Mining Annotation Usage Rules: A Case Study with MicroProfile

Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi
University of Alberta, Edmonton, Canada
{nuryyev, ajaykum1, nadi}@ualberta.ca

Yee-Kang Chang, Emily Jiang, Vijay Sundaresan
IBM, Markham, Canada
yeekangc@ca.ibm.com, emijiang@uk.ibm.com, vijaysun@ca.ibm.com

***Abstract*—While Application Programming Interfaces (APIs) allow easier reuse of existing functionality, developers might make mistakes in using these APIs (a.k.a. *API misuses*). If an API usage specification exists, then automatically detecting such misuses becomes feasible. Since manually encoding specifications is a tedious process, there has been a lot of research regarding pattern-based specification mining. However, while annotations are widely used in Java enterprise microservices frameworks, most of these pattern-based rule discovery techniques have not considered annotation-based API usage rules. In this industrial case study of MicroProfile, an open-source Java microservices framework developed by IBM and others, we investigate whether the idea of pattern-based discovery of rules can be applied to annotation-based API usages. We find that our pattern-based approach mines 23 candidate rules, among which 4 are fully valid specifications and 8 are partially valid specifications. Overall, our technique mines 12 valid rules, 10 of which are not even documented in the official MicroProfile documentation. To evaluate the usefulness of the mined rules, we scan MicroProfile client projects for violations. We find 100 violations of 5 rules in 16 projects. Our results suggest that the mined rules can be useful in detecting and preventing annotation-based API misuses.**

***Index Terms*—pattern mining, API usages, annotation**

I. Introduction

The microservice architectural style ("*developing a single application as a suite of small services, each running in its own process*") is becoming more popular since it allows independent development and deployment of different parts of a system, as well as flexibility in choosing the technology stack to use[1]. There are various frameworks[2][3] that facilitate the development of microservice-based Java enterprise applications, including MicroProfile [1].

MicroProfile is an open-source Java microservices framework developed by the open source community including IBM, Red Hat, Payara, Tomitribe [1]. It provides essential functionality for the development of microservices, such as fault tolerance where backup microservices are specified if one service becomes unavailable. Client developers use MicroProfile mainly through annotation-based Application Programming Interfaces (APIs). For example, in Listing 1, using the MicroProfile `@Asynchronous` annotation indicates that method `foo()` will be executed asynchronously. However, the top `foo()` will not work as expected, because there is

```
// Incorrect usage! Leads to runtime exception
@Asynchronous
public String foo() { ... }

// Correct usage
@Asynchronous
public CompletionStage<String> foo() { ... }
```

Listing 1: Correct and Incorrect example usage of the MicroProfile `@Asynchronous` annotation [2].

an implicit API usage rule that any method annotated with `@Asynchronous` should return an object of type `Future` or `CompletionStage` [2]. If a method annotated with `@Asynchronous` does not return either types, a run-time exception is thrown. We refer to such incorrect usages of the API as *misuses*. While the above misuse leads to an exception that alerts developers that something went wrong, there are also annotation misuses that lead to silent faulty behavior without any explicit error. For example, the author of Stack Overflow question 53934514 experiences an issue where an HTTP endpoint returns HTTP 200 "OK" instead of HTTP 401 "Unauthorized", because they did not add annotation `@RolesAllowed` on a method (endpoint) that they want to limit access to (administrator vs user). Unfortunately, the Java compiler does not check for either of these API misuses. Thus, it would be useful to have automated tools that can specify and check for correct annotation usage.

Annotations are widely used in microservices as well as other Java applications [3], [4], and they can be misused in various ways [5]. There is existing work on designing specification languages to encode annotation usage rules and developing tools to ensure these rules are respected [6]–[8]. However, these annotation checkers assume that an API expert will write the rules or that these rules are explicitly specified in the documentation. Unfortunately, API usage rules are not always documented and expecting framework developers to exert extra effort to encode them in a new specification language is not practical. Ideally, such rules should be automatically inferred.

A popular way to automatically extract API usage rules is to employ pattern mining. The idea of automatically mining general API usage rules, not specific to annotations, has been extensively investigated in the literature [9]–[11]. The common underlying premise is that frequent API usages indicate rules

[1]Microservices Adoption Report: https://www.oreilly.com/pub/pr/3307
[2]Spring Boot: https://spring.io/projects/spring-boot
[3]Micronaut: https://micronaut.io/

that should be respected. However, these efforts do not take annotation usages into account and focus on verifying API usages with control and data flow relationships, typically within the same method. Different from API usage constraints within a method, existing studies [6], [8] on the manual encoding of annotation usage rules indicate that annotation usages do not have a particular order, and the relationships between annotations typically span multiple code elements and artifacts, including configuration files. Thus, tracking and mining such annotation usage is a challenging task, particularly because there is no obvious code linkage between annotations and their dependent code elements or artifacts that we can leverage.

In this work, we describe our collaboration with IBM to devise techniques that mine usage rules (a.k.a *specifications*) for MicroProfile annotations. Such specifications can later be encoded as static analysis checks to detect API misuses, or specified in the API documentation to prevent misuse. To infer annotation usage rules, we first investigate annotation usage constraints by manually analyzing existing annotation usage rules present in the official MicroProfile documentation [1] and other online resources. Based on our observations, we devise a frequent-itemset based pattern mining approach for mining annotation usage rules, which includes the development of various optimization techniques to improve the accuracy and diversity of the mined usage rules.

Overall, we mine 23 candidate rules from 533 MicroProfile projects. Among the 23 candidate rules, a domain expert from IBM confirms 12 rules for MicroProfile. We compare the confirmed rules with the annotation usage rules mentioned in the official MicroProfile documentation and find that ten of the confirmed rules are not documented. This indicates that our approach is effective in discovering new rules and can be useful in improving documentation, which in turn can be useful in preventing annotation misuse. We also scan the 533 MicroProfile projects for violations of the confirmed rules. We find 100 violations of five mined rules in 16 projects, which indicates that the confirmed rules are useful in detecting annotation-based API misuse. Overall, this paper makes the following main contributions:

- Investigate code facts and relationships that need to be considered for mining annotation usage rules for MicroProfile.
- Adapt the well-established frequent itemset mining algorithm to mine annotation usage rules.
- Develop various optimization techniques to increase diversity of mined rules and decrease their redundancy.
- Evaluate the effectiveness of our approach by mining annotation usage rules from 533 MicroProfile client projects and verifying the mined rules with a domain expert, who is a direct contributor to MicroProfile.
- Evaluate the usefulness of the mined rules by detecting and reporting violations in 533 MicroProfile client projects.
- A discussion of our experiences and potential future work.

We also provide all our scripts and data (excluding proprietary projects) in our online replication package [12].

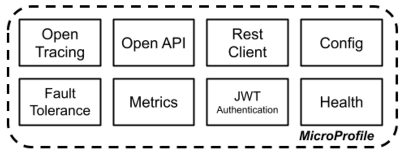


Fig. 1: MicroProfile APIs/components

II. Background

In this section, we first introduce microservices and our target framework, MicroProfile. We also introduce Java annotations and how they are used in MicroProfile. Finally, we explain how frequent itemset mining works, which is the pattern mining algorithm we use in our approach.

A. Microservices and MicroProfile

The *Microservice architectural style* involves building a web application as a set of smaller ("micro") services, each running in a separate process. Microservices have the advantage that they can be independently developed, deployed and scaled.

MicroProfile is an open-source framework that provides a collection of APIs to facilitate the development of microservices in Enterprise Java web applications [1]. MicroProfile provides specifications as separate components, shown in Figure 1, basically decomposing the MicroProfile API based on different aspects of microservices. For example, developers can use the Health component to monitor the health status of their services. A full description of all the components is available on the official website for MicroProfile [1].

B. Java annotations & API usage rules

MicroProfile provides its functionality primarily through Java annotations. *Java annotations* are a form of metadata applied on a variety of language constructs, such as classes, methods, fields, and method parameters[4]. Annotations provide a convenient way of applying additional behavior to annotated constructs, whether at compile or run time.

Consider the `@Asynchronous` annotation from the MicroProfile Fault Tolerance API shown in Listing 1. According to the documentation, this annotation can be applied on a class or method [2]. If `@Asynchronous` is used on a method, then the method will be invoked in a separate thread. Even though using the annotation seems very simple, it comes with an additional usage constraint. As shown in Listing 1, if a method is annotated with `@Asynchronous`, it must return `Future` or `CompletionStage<T>`. Otherwise, a run-time `FaultToleranceDefinitionException` occurs [2].

C. Frequent Itemset Mining

Frequent itemset mining is primarily used for *association rule learning*, a data mining technique for discovering interesting relationships between items in large databases [13].

Consider the following formal definition of frequent itemset mining [14]. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of items and $D = \{t_1, t_2, ..., t_m\}$ be a database of (observed) transactions, where each transaction contains a subset of

[4]https://docs.oracle.com/javase/8/docs/technotes/guides/language/annotations.html

the items in I. Let X be an *itemset*, which is a set of one or more items belonging to I. To find interesting patterns, frequent itemset mining uses the notion of *support*, $supp(X)$, which is the number of transactions an itemset X appears in. For example, given $I = \{bread, jam, eggs\}$, we may have observed the following (purchase) transactions $D = \{\{bread\}, \{eggs\}, \{bread, eggs\}, \{bread, eggs\}, \{bread, jam\}, \{bread, jam, eggs\}\}$. In this case, $supp(\{eggs\}) = 4$ while $supp(\{bread, eggs\}) = 3$. Given a user-provided *support threshold* $supp_{min} \in N$, itemset X is called *frequent* if and only if $supp(X) \geq supp_{min}$. In this example, a support threshold of 3 would produce only {eggs}, {bread}, and {bread, eggs} as frequent itemsets.

Once frequent itemsets are retrieved, they can be used to generate association rules. *Association rules* are relational rules of the form "If Y, then Z", or more precisely $Y \implies Z$ where $Y, Z \subseteq F$ and F is a frequent itemset. The "if" part is called *antecedent*, and the "then" part is called *consequent*. Association rules come with their own indicator called confidence. *Confidence* of a rule $Y \implies Z$ is a measure of how often an association rule $Y \implies Z$ has been found to be true in all the retrieved transactions: conf($Y \implies Z$) = supp($Y \cup Z$) / supp(Y). Following the above example, conf($bread \implies eggs$) is 60% since 3 of the 5 transactions that contain bread also contain eggs, while conf($eggs \implies bread$) is 75%. While the minimum support threshold $supp_{min}$ helps us mine frequent itemsets within a database of transactions, the minimum confidence threshold $conf_{min}$ helps us generate association rules with strong "if-then" relations between items. In other words, an association rule $Y \implies Z$ is generated, if and only if $conf(Y \implies Z) \geq conf_{min}$.

III. Investigating Properties of MicroProfile Annotation-based API Usages

Most of the existing work on mining API usage focuses on data and control flow information [9], [10]. Given the different nature of annotations, we first need to identify what kind of information (*code facts*) we need to track to be able to mine annotation usage rules. Although existing work on the manual encoding of annotation usage rules identify some code facts, the results are not based on a systematic study of annotation usage [6], [8]. Thus, the identified code facts might not be sufficient for MicroProfile annotation usage.

To understand how MicroProfile annotations are meant to be used (or misused), we search for any mention of annotation usage rules in the official MicroProfile documentation [1]. In addition, we search Stack Overflow for any issues client developers have faced while using MicroProfile's API.

To identify existing issues with the usage of MicroProfile annotations on Stack Overflow, we search for posts with tag *microprofile*[5]. As of March 29, 2022, there were 262 questions related to MicroProfile. We look at each post's title, body, comments, and answers to identify any annotation usage rules. For example, in question 53934514, the question author is struggling to secure an HTTP endpoint using JSON Web Tokens (JWT)[6]. They have a field of type `JsonWebToken`, which has access to JWT features. However, when the application is deployed and tested locally, the field has value of `null` and the application does not throw (or even log) any error. The accepted answer to the problem is to use `@RolesAllowed` on the endpoint along with `@DeclareRoles` on the class. Based on the accepted answer, we infer the following rule: *"If an `Application` class is annotated with `@LoginConfig` with parameter value `authMethod="MP-JWT"`, then methods annotated with `@GET` should be annotated with `@RolesAllowed` and the Application class should be annotated with `@DeclareRoles` or the roles must be defined in `web.xml`"*. We present all such inferred rules to our IBM collaborators for confirmation. In this example, they clarify that adding `@DeclareRoles` is not always necessary and there are other role annotations that can also be used depending on the intention. Thus, we modify the extracted rule to: *"If Application class is annotated with `@LoginConfig` with parameter value `authMethod="MP-JWT"`, then a method annotated with `@GET`, or its containing class, must be annotated with `@RolesAllowed`, `@PermitAll`, or `@DenyAll`."*

In total, we extract 12 annotation usage rules for seven different MicroProfile annotations (available on our artifact page). Among the 12 rules, 11 are from the official documentation and one is from Stack Overflow. Based on these annotation usage rules, we identify three code elements that are involved in annotation usage: annotations, program elements (e.g., method return and field types), and configuration files. Among the 12 rules, two rules contain only annotations, one rule contains annotation and configuration files, and the remaining nine rules contain annotations and other program elements.

Based on the 12 manually extracted rules, we identify the following relationships that we would need to track:

- *annotatedWith* to track an annotation on a program element. We observe the following program elements: class, field, method, constructor and method/constructor parameters.
- *hasType* to track the type of a field.
- *hasParam* to track a parameter type of some program element, including annotation parameters.
- *hasReturnType* to track a method's return type.
- *extends* to track class extensions of MicroProfile classes.
- *implements* to track a class implementations of MicroProfile interfaces.
- *definedIn* to track the connection between an annotation parameter value and a key defined in `microprofile-config.properties` configuration file.
- *declaredInBeans* to connect a class to the `beans.xml` file that typically contains class configurations.

IV. Mining MicroProfile Annotation Usages

Figure 2 shows a high-level overview of our approach for mining annotation usage patterns. Based on the relationships

[5]https://stackoverflow.com/questions/tagged/microprofile

[6]https://jwt.io/

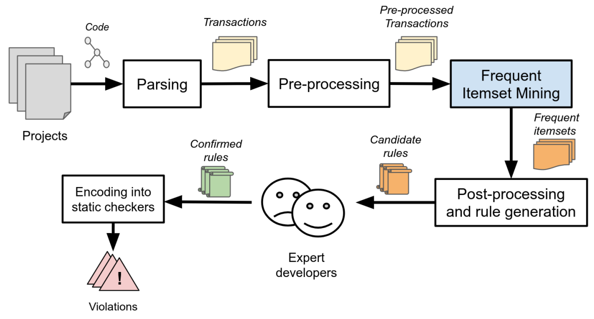


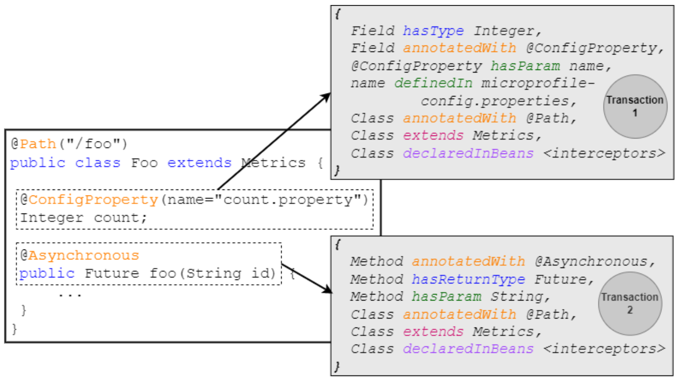
Fig. 2: Overview of our mining and validation approach.

we identified in Section III, we first parse client projects to extract transactions, where each item in a transaction corresponds to one of these relationships (Section IV-A). We then pre-process the extracted transactions to ensure that we mine as many diverse annotation usage patterns as possible (Section IV-B). We then use frequent itemset mining to mine usage patterns as frequent itemsets (Section IV-C). Finally, we post-process the frequent itemsets to remove invalid patterns and generate candidate rules in an "if-then" form (Section IV-D). The expert validation of candidate rules and the encoding of confirmed rules are part of our evaluation setup (Section V).

A. Parsing Input Client Projects to Extract Transactions

We use JavaParser to parse client projects and extract transactions that represent annotation usage examples from client code [15]. Figure 3 depicts an example of a code snippet and configuration files, as well as the relationships that we extract from this example. In Figure 3a, there are three entities that we keep track of: class, field, and method. For the purposes of frequent itemset mining, we store the code facts of each entity in a separate transaction. In other words, in our context, an item is a relationship that tracks some code fact and a transaction is a usage example with a list of observed relationships. In the figure, we have the "Transaction 1" area that contains a transaction that represents the field `count` and the "Transaction 2" area that contains a transaction that represents the method `foo`. For example, in "Transaction 2", we extract the relationship "*Method hasParam String*", which means that the method has a parameter of type `String`. Note that we do not create a separate transaction for the class `Foo` and instead put all relevant class information into all transactions that represent the class members such as its fields and methods. The reason for this decision is that creating a separate transaction for a class would not allow us to mine potential rules that involve relationships between classes and fields or classes and methods. On the other hand, we do not encode information of multiple members of a class together in a transaction, because it will be impossible to track which information belongs to which member of a class (e.g., encoding two different methods in the same transaction would not differentiate their parameters).

The above process produces a list of transactions from client projects. In the next step, we proceed to pre-processing these transactions before we mine frequent itemsets from them.

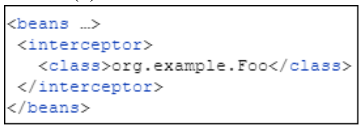


```
@Path("/foo")
public class Foo extends Metrics {

 @ConfigProperty(name="count.property")
 Integer count;

 @Asynchronous
 public Future foo(String id) {
     ...
 }
}
```

```
{
  Field hasType Integer,
  Field annotatedWith @ConfigProperty,
  @ConfigProperty hasParam name,
  name definedIn microprofile-
          config.properties,
  Class annotatedWith @Path,
  Class extends Metrics,
  Class declaredInBeans <interceptors>
}
```

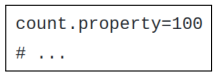
```
{
  Method annotatedWith @Asynchronous,
  Method hasReturnType Future,
  Method hasParam String,
  Class annotatedWith @Path,
  Class extends Metrics,
  Class declaredInBeans <interceptors>
}
```

(a) Extracted transactions for an example code snippet



```
<beans ...>
 <interceptor>
   <class>org.example.Foo</class>
 </interceptor>
</beans>
```

(b) Example `beans.xml` file



```
count.property=100
# ...
```

(c) Example `microprofile-config.properties` file

Fig. 3: Parsing client projects into transactions based on supported relationships

B. Pre-processing Transactions

Since API usage distribution might be skewed in client projects, after we convert usages into transactions, we pre-process the transactions by normalizing the API usage distribution before mining patterns from them.

Take a set of unique transactions from each project: Frequent itemset mining is likely to yield common usages (idioms), while our goal is to ideally mine only usage patterns that represent usage specifications. One of the reasons for mining common idioms is skewed input data. One project might have thousands of usages that are the same, while other projects have insignificant to no amount of such usages. To reduce the bias towards few projects, we take a set of unique transactions per project. For example, if project *A* has a list of transactions `[T1, T2, T3]`, where `T1` and `T2` are identical (i.e., have the same items), we keep `T1` and delete `T2`, ending up with `[T1, T3]`. Thus, our rule mining prioritizes usages that appear in multiple projects rather than just one project.

Partition transactions by annotations: The MicroProfile framework spreads its functionality into different components (represented as different packages), such as Rest Client and Fault Tolerance, as shown in Figure 1. We observe that there is an imbalance in terms of frequency of usage of different packages and their annotations. For example, the MicroProfile Rest Client (i.e., `org.eclipse.microprofile.rest.client`) package is used frequently, whereas MicroProfile Reactive (i.e., `org.eclipse.microprofile.reactive`) is used rarely. We also observe the same frequency differences for the annotations within these packages; there is only a small fraction of annotations within each package that is frequently used by client developers. For example, within the MicroProfile Rest Client package, `@RegisterRestClient` is used frequently, whereas `@RestClientBuilder` is used rarely. Overall, these observations indicate that using a fixed support threshold to mine annotation usage rules for all MicroProfile annotations would not work.

Therefore, to increase the likelihood of mining rules from not-so-popular annotations, we partition the list of all transactions mined from the projects into smaller, separate lists for each annotation. So if a transaction has multiple MicroProfile annotations, the transaction will appear in multiple transaction lists. We mine frequent itemsets from each of these transaction lists separately in the next step.

C. Mining Frequent Itemsets

We leverage the parallelized version of the popular mining algorithm *FP-Growth* [16], [17] to mine frequent itemsets. FP-Growth constructs a prefix tree *FP-Tree* of transactions, from which only frequent itemsets are generated. However, frequent itsemset mining usually produces a large number of frequent itemsets that are not very useful or too similar to each other. For example, one frequent itemset can be a superset of several others. We therefore post-process frequent itemsets before generating any candidate usage rules.

1) Optimizing Frequent Itemsets: Based on our observations as well as discussions with our IBM collaborators, we develop a set of optimization techniques that help us focus on the most relevant frequent itemsets and remove irrelevant ones. These heuristics have to be applied in the described order.

Remove redundant frequent itemsets: To reduce the number of similar frequent itemsets, we keep only *maximal* frequent itemsets, i.e., frequent itemsets that have no proper superset. For example, say we mine two frequent itemsets $A, B \in F$, where F is the set of all frequent itemsets, and $A = \{a, b\}$ and $B = \{a, b, c\}$. Since B is a superset of A by inclusion ($B \supseteq A$) and already includes all the information contained within A, we keep only B and remove A.

Remove frequent itemsets with no target API usage: Developers tend to use different libraries and frameworks in their code to accomplish different tasks. Our technique focuses on mining API usages of MicroProfile. Therefore, we are not interested in API usages from other libraries or frameworks. Thus, we remove all frequent itemsets that do not have at least one element of MicroProfile API.

Remove semantically incorrect frequent itemsets: A frequent itemset must contain relationships that form a semantically correct sequence of actions for a given usage. Otherwise, a frequent itemset will result in meaningless association rules. We identify two conditions, both related to annotation parameters, for semantically incorrect frequent itemsets. First, a frequent itemset has a *"@A hasParam P"* relationship, but not a *"... annotatedWith @A"* relationship. Second, a frequent itemset has a *"P definedIn Config"* relationship, but not a *"@A hasParam P"* relationship. Basically, recording that an annotation has a parameter without the presence of the annotation on a program element in the first place is meaningless. Similarly, recording that an annotation parameter is present in a configuration file without the presence of this parameter as part of an annotation is meaningless. Note that while transactions always capture semantically correct code, and thus these conditions always hold, the same is not true for frequent itemsets, which is why we need this filtering step. For example, given transactions T_1 = {*"**class** annotatedWith @A", "@A hasParam x"*} and T_2 = {*"**method** annotatedWith @A", "@A hasParam x"*} and minimum support 2, we will get a frequent itemset F = {*"@A hasParam x"*}. However, this frequent itemset is meaningless and does not represent a semantically valid usage. Thus, we exclude any frequent itemsets that violate the above two conditions.

D. Generating Candidate Rules

We generate association rules from the remaining semantically valid maximal frequent itemsets with MicroProfile API usages from Section IV-C. We refer to these association rules as *candidate API usage rules* (or *candidate rules* for short), because they represent *potential* usage rules, but we cannot be certain until they have been verified by an API expert.

Association rule mining generates candidate rules in the form of *antecedent* and *consequent*. Given a frequent itemset F = {X, Y}, association rule mining will generate rules R_1 = {X $\Longrightarrow$ Y} and R_2 = {Y $\Longrightarrow$ X} as long as the minimum confidence threshold is met, regardless of whether the semantics of an implication in that direction makes sense or not. Based on our earlier manual extraction of annotation rules (Section III), we observe two types of implicit implication directions related to annotations that we leverage to modify the order of likely incorrect implications. First, the presence of an annotation may imply the need for other related program elements (e.g., method return and field types), not the other way around. For example, a field annotated with `@A` may imply that this field must have a specific type `T`. However, it is unlikely that the fact that a field is of a specific type `T` implies that the field must be annotated with a specific annotation. Second, the presence of method and field-level annotations can imply the need for specific class-level annotations, but the other way round is unlikely. Thus, we check if a generated candidate rule contains an unlikely implication direction and change the order of the implication to increase the chances of this candidate rule being correct.

V. Evaluation

To evaluate the effectiveness of our approach and usefulness of the mined candidate rules, we investigate the following reasearch questions:

- **RQ1: How effective is our pattern mining approach in discovering annotation usage rules?** We investigate whether our approach can mine correct annotation usage rules. Since the mined candidate rules may not exactly match a correct usage rule (e.g., have extra or missing relationships), we investigate how much the candidate rules have to be edited to become correct rules. We also investigate whether our approach can mine *known rules* (manually extracted in Section III) and *new rules* (not documented).
- **RQ2: How common are violations of mined annotation usage rules?** To demonstrate the potential usefulness of the

mined and confirmed usage rules, we encode them into static checkers and search for violations of these rules in projects.

A. RQ1: Effectiveness of our approach

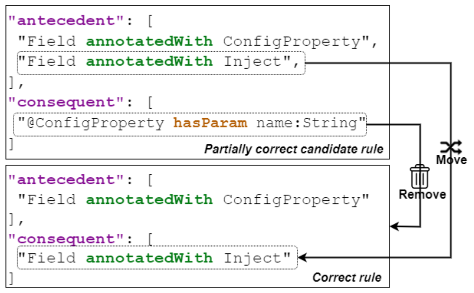
1) Evaluation Setup: To evaluate our approach, we retrieve client projects that use MicroProfile using its GitHub repository's dependency graph. We filter out toy projects and focus only on those repositories whose title does not contain any of the following keywords: "demo", "example", "playground", "getting-started", "sample", "starter", "quickstart", "quick-start", "tutorial"; and whose sizes are more than 500 KB. We do not filter out projects by stars, because we find that the majority of the projects have no stars due to the framework being relatively new (first released in 2016 [1]) and the tendency of the client projects to be closed-source. To address the latter issue, we also clone 81 proprietary closed-source projects from the IBM GitHub organization. In total, we analyze 533 MicroProfile projects.

2) Method: We follow the methods we described in Section IV. After parsing the client projects into transactions and pre-processing the transactions, we mine frequent itemsets per API. We set the *relative* minimum support threshold to 80%, which means that an itemset is frequent if and only if it appears in at least 80% of transactions. We choose 80% threshold, because our initial experiments showed that values lower than that lead to too many frequent itemsets being generated without gain of any new information, while higher values lead to very few frequent itemsets. We then generate candidate rules from the frequent itemsets with an 85% confidence threshold. Since the candidate rules may not necessarily represent correct MicroProfile API usage rules, we present our candidate rules to a domain expert, who is one of our IBM collaborators and a direct contributor to MicroProfile.

Precision and Recall: We ask the domain expert to confirm or deny whether the candidate rules represent correct rules without any modifications. Based on the results, we measure precision and recall. We measure *precision* as the number of confirmed rules among the mined rules: $R_{confirmed}$ / R_{mined}. In other words, how many of the mined rules are correct rules. Whereas, we measure *recall* as the number of confirmed rules among the know rules we manually extracted in Section III: $R_{confirmed}$ / R_{manual}. In other words, how many of the known rules were we able to mine.

Edit Distance: The rejected candidate rules do not represent correct rules in their currently mined form; however, they might represent correct rules with some editing. Therefore, we ask the domain expert to mark the rejected candidate rules as partially correct rules if they can be edited into correct rules and to record these edits. The editing process involves the following operations:

- **Remove** a relationship from a candidate rule.
- **Move** a relationship from the antecedent to consequent, or vice versa.
- **Add** a new relationship to a candidate rule. A new relationship can be added independently or disjointly (i.e. join with an existing relationship using "OR").



```
"antecedent": [
 "Field annotatedWith ConfigProperty",
 "Field annotatedWith Inject",
],
"consequent": [
 "@ConfigProperty hasParam name:String"
]
```

```
"antecedent": [
 "Field annotatedWith ConfigProperty"
],
"consequent": [
 "Field annotatedWith Inject"
]
```

Fig. 4: Editing a partially correct rule into a correct rule.

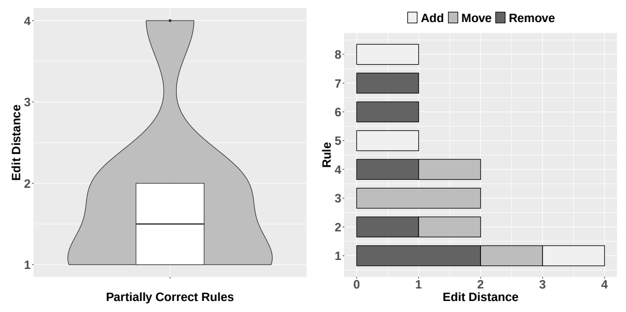
Using the three operations above, we calculate the edit distance between a candidate rule and its corresponding correct rule. Assuming that each edit operation costs one unit, the *edit distance* is the sum of all the edit operations suggested by the expert for a candidate rule. For example, the edit distance between the candidate and correct rule in Figure 4 is two, because it requires two edit operations (one **Remove** and one **Move**). Ideally, we want the edit distance of a candidate rule to be as close to 0 as possible, implying that minimal (or no) effort is needed to convert a candidate rule into a correct one.

Newly Discovered Rules: After the editing process, we collect all accepted rules, which include the confirmed rules and the converted fully correct rules. We then check if these accepted rules contain any new rules we did not previously observe. The *new rules* are those rules that are not already part of our manually extracted rules (i.e. not already *known rules*). Ideally, we want to mine both *known rules* and *new rules*, but mining new rules is more significant, because it indicates that the code facts we discovered in Section III are not only limited to mining the known rules but also suitable for mining new annotation usage rules. In fact, the whole idea of our approach is to discover new rules based on limited pre-existing knowledge.

In summary, we use precision, recall, edit distance, and new rules to measure the effectiveness of our approach.

3) Results: We mine 23 candidate rules for MicroProfile. Based on manual expert validation, we find that four out of the 23 candidate rules are correct rules that do not require any edits, which means that we mine rules with 17% precision. We also find that none of the four correct rules are known rules (recall 0%) and that they are all new rules.

Among the remaining 19 candidate rules, our domain expert identifies eight candidate rules as partially correct, which means they need some editing to be turned into correct usage rules (full list on our artifact [12]). The remaining 11 candidate rules do not correspond or provide starting points to any real rules. Figure 5a shows the edit distance distribution for the eight partially correct rules. This figure indicates that a domain expert who is using these partially correct candidates as a starting point to create usage rules/specifications would have to perform 1.75 edit operations on average (median = 1.5). Note that the average size of these eight rules is 2.75 items (median = 3). The previously discussed Figure 4 is an example of a real rule that we mine and confirm with the domain expert



(a) Edit distance distribution (b) Edit operation per rule

Fig. 5: Edit distance distribution of partially correct rules

that needs two edit operations.

Figure 5b shows the distribution of needed edit operations in each of the eight rules. The **Remove** operation is the most frequent edit operation (43%), followed by the **Move** operation (36%). The least needed operation is the **Add** operation (21%) implying that our pattern mining approach captures most of the items necessary to make up an actual rule. Out of the eight partially correct rules, five rules require only reshuffling and removal of some items to be turned into correct rules.

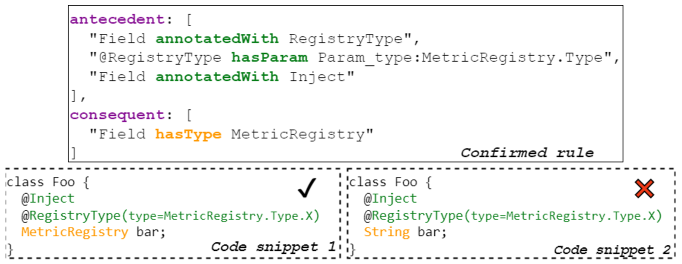
Overall, we mine four correct rules and eight partially correct rules. This means that overall we have 12 confirmed rules that our mining technique was able to mine as is or provide a starting point for. Among these 12 rules, six rules contain only annotations, four rules contain annotations and program elements, and the remaining two rules contain annotations and configuration files. We find that two of the 12 final rules are known rules. The remaining 10 rules for 10 different MicroProfile annotations are new rules, which means these rules are not documented in the official MicroProfile documentation.

Answer to RQ1: Our pattern mining approach finds 12 confirmed rules for MicroProfile, eight of them require some editing to be turned into actual rules. Among the 12 confirmed rules, 10 are new rules that are not documented in the official MicroProfile documentation.

B. RQ2: Usefulness of the mined rules in detecting violations

1) Evaluation Setup: To find violations of the confirmed usage rules, we analyze real-world client projects that use MicroProfile. We use the same set of 533 projects that we use for mining candidate rules, mainly due to unavailability of additional non-toy projects.

2) Method: We develop a simple static analysis tool based on JavaParser to analyze code for violations. We first encode checks for the 12 confirmed MicroProfile rules in our static analysis tool. To find violations of a rule, our tool searches for the relevant code element where the rule antecedent is `true` but the consequent is `false`. For example, Figure 6 depicts a correct usage (code snippet 1) and a violation (code snippet 2) of one of the encoded MicroProfile rules. Given this analysis, we then run our tool on each project's latest commit to detect violations of any of the encoded rules. We also run the analysis



```
antecedent: [
  "Field annotatedWith RegistryType",
  "@RegistryType hasParam Param_type:MetricRegistry.Type",
  "Field annotatedWith Inject"
],
consequent: [
  "Field hasType MetricRegistry"
]
                                        Confirmed rule
```

```
class Foo {                                        ✓
  @Inject
  @RegistryType(type=MetricRegistry.Type.X)
  MetricRegistry bar;
}                                   Code snippet 1
```

```
class Foo {                                        ✗
  @Inject
  @RegistryType(type=MetricRegistry.Type.X)
  String bar;
}                                   Code snippet 2
```

Fig. 6: Encoded rule with correct and incorrect usage.

on all commits from the projects' commit history. Overall, our goal is to find any real violations of these rules.

3) Results: We find 100 MicroProfile API usage violations of five distinct usage rules in 16 client projects. Among the 100 violations, 48 violations are in the commit history of 10 projects, which have been already fixed by developers. Among the 48 violations, 14 violations have been fixed by adding the missing *consequent*, 23 violations have been fixed by removing *antecedents*, and the Java files containing *antecedents* have been removed for the remaining 11 violations. Apart from the 48 violations in the commit history, we find 52 violations in the latest commit of eight projects. We submitted a pull request (PR) per project to fix these 52 violations. To date, two of these PRs which fixes three violations of three different rules in two projects have been accepted [12]. The violations we detect have two types of consequences: runtime exceptions and silent, faulty behavior which does not result in any explicit error. For example, in both open-source and proprietary projects, we find violations of usage rule "*If a method is annotated with **@Outgoing**, then the class containing the method should be annotated with **@ApplicationScoped** or **@Dependent**, or there should be* `beans.xml` *file in the project*". In the violation, the developer uses the *@Outgoing* annotation, but does neither add the other annotations nor have the `beans.xml` file in the project. The violation leads to silent, faulty behavior, because the target class is ignored by the runtime deeming the *@Outgoing* annotation useless.

Answer to RQ2: We discover 100 violations of five MicroProfile rules in 16 projects. We submitted eight PRs for 52 of the 100 violations that were not already fixed by developers. Among the eight PRs, two PRs for three violations have been accepted so far, which indicates that the violations are real.

C. Threats to Validity

Internal Validity. We set the support and confidence thresholds based on our initial experiments, because there is no precedent for setting the thresholds for mining annotation usage rules. To ensure the correctness of our miner and static analysis checkers, we created a set of synthetic (test) code snippets that we use to mine relationships and verify whether the mined set of relationships is equal to the expected one.

Construct Validity. We use edit distance as a proxy for how much effort the API experts need to exert in the process of confirming a final rule. However, an edit distance of four is not necessarily "twice as bad" as an edit distance of two, because not all edit operations are equally intellectually demanding. For example, according to feedback from our industrial collaborators, adding a missing relationship is harder than moving or removing a relationship that is already present in the rule. Thus, the edit distance we use is a simple proxy for the effectiveness of the mined rules. More precise measures may include (1) a longitudinal study where experts confirm even more rules over time and where we measure the time it takes them to decide about a rule, what the final rule looks like, and additionally gather their feedback about the difficulty of editing rules or (2) a controlled experiment where one group of experts authors rules from scratch while the others use our mined candidate rules as their starting points. In this paper, we use only one expert to modify and validate the mined rules. A different expert may consider a partially correct rule as not a rule or modify a partially correct rule into a different correct rule. However, it is highly unlikely that any of the 12 confirmed rules is an invalid rule. We also found that developers have fixed violations of five of these confirmed rules, further confirming their correctness.

External Validity. While several frameworks provide functionality for building microservices, we focus only on MicroProfile APIs driven by a real industrial need. Therefore, we do not know if our results generalize to other frameworks, which can be part of future work. That said, apart from the configuration-related code facts, all other code facts our approach supports for mining annotation usage rules apply to all Java-based frameworks. Therefore, we believe our approach can be easily extended to mine annotation usage rules for other frameworks by adding support for tracking framework-specific configuration-related code facts. The majority of the client projects we analyze are open-source projects. Although we filtered projects to ensure that we analyze only high-quality projects, the projects might not represent real-world industrial projects. Therefore, we also added 81 closed-source IBM projects that use MicroProfile.

VI. Discussion

The motivation of conducting this research stemmed from a real industry problem of ensuring that developers correctly use annotations, specifically in MicroProfile. In this work, we investigated one simple pattern mining approach for discovering rules and its application in practice. We now discuss the implications of our findings from this experience.

a) Potential for Rule Authoring: Our approach mines 4 fully correct and 8 partially correct rules. We also find that it takes 1.75 edits, on average, to convert a partially correct rule into a fully correct one. Overall, our approach mines 12 confirmed annotation usage rules. The results indicate that using simple frequent itemset mining techniques is promising for extracting annotation-based API usage rules. While 8 of the 12 confirmed mined rules require editing to become complete usage specifications, we find that the majority of these edits are reshuffling and removal of some items. Based on domain experts' feedback, it is easier to reshuffle or remove an item rather than try to identify a missing one and add it to a rule. Thus, our pattern mining approach provides an effective starting point for framework developers to create API usage rules, which avoids the need to write all specifications manually from scratch. We are currently working on providing more streamlined tools for MicroProfile developers to easily confirm, reject, or modify mined rules.

b) Rule Coverage: We find 12 usage rules for MicroProfile, which shows the promise of this mining approach. However, our approach could mine only two of the 12 manually extracted rules (i.e. known rules), suggesting that the approach has limitations in terms of the types of rules it can mine. For example, we are not able to mine rules that involve *relationships across members of a Java class*, such as across two methods. For example, Listing 2 shows two methods, where `getEntry2` is a *fallback* method for `getEntry1` method. The rule for using MicroProfile `@Fallback` specifies that the value of annotation parameter, `getEntry2` in this example, should be a method that exists within the same class (or class hierarchy) and should have the same method signature (i.e., the same types of parameters and return type). We are not able to mine such usage rules, because we track relationships for one entity at a time (method, class, field, or constructor). This is an inherent limitation of frequent itemset mining, which relies on transactions with sets of items. Each transaction must have only unique items. To address this limitation, a possible direction is to employ a more advanced representation such as graph. In a graph, nodes may represent the entities (e.g., a method) and edges may represent the relationships on a given entity. However, while the graph representation is likely to capture more complex rules than the ones we mine, the above rule is unlikely to be automatically extracted due to its specificity. In other words, it is hard, if not impossible, for pattern mining to infer that two methods' signature (i.e., return type and parameter list types) should be the same, unless we explicitly encode a relationship that represents this knowledge. However, encoding a relationship for each specific corner case beats the purpose of inferring new knowledge through pattern mining. Finding the right balance is an open problem [10], [18], which often requires domain-specific solutions.

We also find that frequent itemset mining has limitations in mining the rules that have *optional relationships*. For example, take the rule from Listing 1, any method annotated with `@Asynchronous` should return an object of type `Future` or `CompletionStage`. Since client developers could use `Future` or `CompletionStage` in their code, it is not guaranteed that either of them would appear frequently (i.e. above the minimum support threshold) with `@Asynchronous` in transactions. Even if one of them appears frequently (e.g, 85% of the time), the other is guaranteed to not appear frequently (15%) due to mutually exclusive relationships unless the support threshold is $\leq$50%. So either the approach will not be able to mine such rules, or it will mine rules with only one of the optional relationships. This is the reason we had to **add** another optional relationship disjointly in three different partially correct rules during the editing process (see Section V-A3). Although using a low support threshold

```
public class Foo {
    // Call `getEntry2()` if `getEntry1()` fails
    @Fallback(fallbackMethod="getEntry2")
    public String getEntry1(String id) { ... }

    public String getEntry2(String id) { ... }
}
```

Listing 2: Specifying a fallback method

($\leq$50%) might preserve optional relationships in candidate rules, the approach would generate a large number of useless and redundant rules. One potential solution is to infer optional relationships from annotation declarations and add them in candidate rules. For example, if `@A` can be used on both class and method, e.g. "Class *annotatedWith* `@A`" and "Method *annotatedWith* `@A`", and candidate rules have only one of the optional relationships, we can add the other one in the candidate rules. However, we cannot use this technique to infer optional relationships for the other types of program elements, such as method return and field types. In general, identifying optional non-frequent relationships or patterns is a challenging task [19].

c) Value of rule mining: The real value of our approach comes from its ability to infer new knowledge, which is the discovery of new rules in our case. While our approach has some limitations in terms of the types of rules it can mine, we are still able to discover 10 new rules for MicroProfile. These rules are not even documented in the official MicroProfile documentation, which is a key source for client developers to understand the usage of MicroProfile APIs. Note that we found only 11 out of the 12 manually extracted rules in the official documentation, which means our approach generates almost the same number of new rules as the number of pre-existing rules we knew about. The new rules are also diverse in terms of the coverage of different MicroProfile annotations. MicroProfile framework developers can use these new rules to improve the official documentation, which in turn can help client developers in preventing potential API misuse.

d) Employing Specification Checking: When evaluating the impact of the confirmed mined rules, we find 100 MicroProfile API usage violations of five unique rules in 16 real-world client projects. While these results suggest that violations of some of our confirmed mined rules exist (and that some rules are more likely to get violated), we do not find violations of all the encoded rules. This may imply that violations of these rules do not make it to committed code, which suggests that providing tooling that detects these violations more locally during development, e.g., in the IDE, may be more effective for supporting developers.

VII. Related Work

A. Writing API Usage Specifications

Automated static checkers can analyze source code to detect incorrect API usages. However, these checkers need specifications that describe the correct usage. There is a plethora of Domain Specific Languages (DSLs) [20] for writing API usage specifications, often for a particular type of library, such as CrySL for usages of Java cryptography libraries [21] or uContracts for coding idioms and naming conventions for Java projects [22]. The written specifications are then compiled into static analysis checkers that automatically scan for violations.

However, to the best of our knowledge, these tools do not support specifying metadata such as Java annotations. There are DSLs specifically designed for writing annotation-based usage specifications, such as Annabot [8], RSL [6], and a tool based on XQuery [23]. These DSLs provide logical quantifiers such as "for all" and "there exists", as well as logical relations "AND", "OR", etc. to specify relations between annotations and other program elements (e.g., return type). Some existing techniques also provide custom Java annotations that are used to add extra semantics on top of annotation declarations [24], [25]. These custom annotations are then pre-processed to verify correctness of annotations usages. The main problem with all the above tools is that one has to first learn the specification language and then write the API usage specifications manually, which is difficult and time-consuming, which is why using data (pattern) mining techniques to extract API usage specifications automatically is often used [10].

B. Mining API Usage Specifications

The general assumption behind mining API usage specifications is that when dealing with massive amounts of client code that uses some target API, the majority of *repeated* usages are correct. Thus, data mining techniques can be used to find *usage patterns*, i.e., parts of usages that commonly occur within many usage examples. Researchers then treat these patterns as usage specifications that are used to verify whether some API usage is correct or incorrect [9].

Patterns can take a variety of forms such as unordered and sequential [10]. There are also representations and techniques that can capture more complex usages (e.g., data and control flow) such as frequent sub-graph mining [9], [26], [27]. Annotations, on the other hand, do not have complex (data and control flow) semantics; it typically only matters whether you apply a certain annotation, often with particular parameter values. While graphs can potentially be used to capture annotation usage patterns, graphs are complex and mining them is computationally expensive. In our work, we also consider non-code artifacts, such as configuration files. Some existing approaches do mine usages from related artifacts, such as configuration files [28] and code comments [28], [29]. However, these approaches are not based on *pattern* mining, but rather rely on regular expressions or parsing heuristics (e.g., looking for pre-defined words in code comments).

To the best of our knowledge, while there is a multitude of tools that leverage pattern mining techniques to mine API usage specifications from code [10], [30], [31], none of them mine annotation usage specifications. Liu et al. [32] recently proposed a deep learning based model, trained on structural (abstract syntax trees) and textual (tokens) contexts of source code, to recommend Java annotations and detect their misuse. However, the approach cannot provide precise reasons (i.e.

antecedents) for recommending annotations, because the approach does not identify which specific code element, among all the considered code (the source code of entire class or method), is used to recommend annotations. Consequently, the approach cannot be used to extract rules for annotation usage since we do not know the precondition (i.e., the antecedent in our case) that is required to apply this rule. Moreover, the approach cannot detect annotation misuses that are not related to the annotation itself but are related to other relevant (annotation-dependent) program elements, such as the misuse example shown in Figure 6, where the type of the field is incorrect. Our manually extracted rules and confirmed mined rules show that annotation-dependent program elements are critical components of MicroProfile annotation usage rules.

VIII. Conclusion

In this paper, we present an industrial case study of mining annotation usage rules of the enterprise microservices framework, MicroProfile. Through our approach, we were able to mine 12 confirmed MicroProfile usage rules, 10 of which are not even documented in the official MicroProfile documentation. We find 100 violations of five rules in 16 MicroProfile projects. The violations may lead to run-time exceptions or worse, silent faulty behaviors that are hard to debug, which shows the usefulness of the mined rules. Our results show the potential of pattern mining as a starting point for automatically creating annotation usage specifications for MicroProfile as well as the need for tools that could assist client developers in using these specifications to detect and prevent annotation misuse.

Acknowledgments

This research was funded by the IBM Center for Advanced Studies (CAS). Also, thanks to Karim Ali for feedback on various stages of this work.

References

[1] MicroProfile, "Optimizing Enterprise Java for a Microservices Architecture," https://microprofile.io/, 2021, [accessed 14-March-2022].

[2] Emily Jiang, "Asynchronous," https://download.eclipse.org/microprofile/microprofile-fault-tolerance-3.0/apidocs/org/eclipse/microprofile/faulttolerance/Asynchronous.html, 2020, [accessed 14-March-2022].

[3] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the usage, evolution and impact of java annotations in practice," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 969–986, 2019.

[4] A. K. Jha and S. Nadi, "Annotation practices in android apps," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 132–142.

[5] P. Pinheiro, J. C. Viana, M. Ribeiro, L. Fernandes, F. Ferrari, R. Gheyi, and B. Fonseca, "Mutating code annotations: An empirical evaluation on java and c# programs," *Science of Computer Programming*, vol. 191, p. 102418, 2020.

[6] Y. Zhang, "Checking metadata usage for enterprise applications," Ph.D. dissertation, Virginia Tech, 2021.

[7] M. Fähndrich, "Static verification for code contracts," in *International Static Analysis Symposium*. Springer, 2010, pp. 2–5.

[8] I. Darwin, "Annabot: A static verifier for java annotation usage," *Advances in Software Engineering*, vol. 2010, 2009.

[9] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static api-misuse detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 265–275.

[10] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2012.

[11] S. Nielebock, R. Heumüller, K. M. Schott, and F. Ortmeier, "Guided pattern mining for api misuse detection by change-based code analysis," *Automated Software Engineering*, vol. 28, no. 2, pp. 1–48, 2021.

[12] UAlberta SMR Research Group, "Mining annotation usage rules," https://github.com/ualberta-smr/MiningAnnotationUsageRules, 2022.

[13] G. Piatetsky-Shapiro, "Discovery, analysis, and presentation of strong rules," *Knowledge discovery in databases*, pp. 229–238, 1991.

[14] C. Borgelt, "Frequent item set mining," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 2, no. 6, pp. 437–456, 2012.

[15] JavaParser.org, "JavaParser - Home," http://javaparser.org/, 2022, [accessed 16-March-2022].

[16] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[17] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, 2008, pp. 107–114.

[18] S. P. Jashma, D. Acharya, and N. S. Reddy, "A list based redundancy removal approach by mining closed and non-derivable frequent itemsets," in *2020 4th International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 2020, pp. 52–58.

[19] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 283–294.

[20] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[21] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2019.

[22] A. Lozano, K. Mens, and A. Kellens, "Usage contracts: Offering immediate feedback on violations of structural source-code regularities," *Science of Computer Programming*, vol. 105, pp. 73–91, 2015.

[23] M. Eichberg, T. Schäfer, and M. Mezini, "Using annotations to check structural properties of classes," in *International Conference on Fundamental Approaches to Software Engineering*, 2005, pp. 237–252.

[24] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt, "Co-evolving annotations and source code through smart annotations," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 117–126.

[25] J. L. de Siqueira, F. F. Silveira, and E. M. Guerra, "An approach for code annotation validation with metadata location transparency," in *International Conference on Computational Science and Its Applications*. Springer, 2016, pp. 422–438.

[26] H. J. Kang and D. Lo, "Active learning of discriminative subgraph patterns for api misuse detection," *IEEE Transactions on Software Engineering*, 2021.

[27] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu, "Marble: Mining for boilerplate code to identify api usability problems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 615–627.

[28] C. Wen, Y. Zhang, X. He, and N. Meng, "Inferring and applying def-use like configuration couplings in deployment descriptors," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 672–683.

[29] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 242–253.

[30] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[31] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, 2009, pp. 383–392.

[32] Y. Liu, Y. Yan, C. Sha, X. Peng, B. Chen, and C. Wang, "Deepanna: Deep learning based java annotation recommendation and misuse detection," in *2022 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022.