Mining Annotation Usage Rules of Enterprise Java Frameworks

by

Batyr Nuryyev

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Application Programming Interfaces (APIs) allow developers to reuse existing functionality without knowing the implementation details. However, developers might make mistakes in using APIs, which are known as API misuses. One way to detect and prevent API misuses is to encode usage specifications in static checkers that automatically verify the correctness of API usages. There are two popular approaches to encoding specifications: manual encoding by API developers and automatic mining of the specifications from the source code. Manual encoding of specifications is a tedious process, and it also requires that specifications are known in advance. On the other hand, encoding specifications by mining source code is more efficient and requires no advanced knowledge of specifications. However, existing mining tools do not extract annotation-based API usage rules, while annotations are widely used in enterprise microservices Java frameworks.

In this thesis, we investigate whether the idea of pattern-based discovery of rules can be applied to annotation-based API usages for enterprise microservice frameworks. We evaluate the effectiveness and usefulness of our approach on two different microservice frameworks: MicroProfile and Spring Boot. We select MicroProfile based on our industry partner's interests while we select Spring Boot, which is a widely used microservice framework, for generalizability. Our approach successfully mines 73 annotation-based API usage rules. We verify the correctness of the mined rules with expert developers. We find that the mined rules required some edits to become fully correct, which is much better than writing rules manually from scratch. To evaluate the usefulness of the mined rules, we scan more than 1,500 client projects that use MicroProfile or Spring Boot frameworks for rule violations. We find 28 violations in 24 projects, which we report as issues in GitHub. We also analyze commit histories of the projects to check whether developers have made mistakes in using annotation-based APIs corresponding to the mined rules and fixed them. We find 12 violations in 9 projects that have been fixed by developers. Overall, the results show that the mined rules can be useful in detecting and preventing annotation-based API misuses.

Preface

We intend to submit Chapters 4, 5, and 6 of this thesis for a publication.

We can only see a short distance ahead, but we can see plenty there that needs to be done.

– Alan Turing, Computing Machinery and Intelligence, 1950.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Sarah Nadi, for her continuous guidance and support throughout my graduate studies. She has been an exemplary mentor whose feedback pushed me to my limits and helped me grow as a researcher.

I would also like to thank our collaborators Emily, Yee-Kang, Vijay from IBM, as well as my colleagues Ajay and Mansur for offering advice and constructive feedback throughout this research.

This research was undertaken, in part, thanks to funding from IBM CAS Canada.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Microservices architectural style is becoming more popular due to independent development and deployment, as well as flexibility in choosing the tech stack. According to O'Reilly's "Microservices Adoption in 2020" report [58], 77% of respondents (which includes software engineers, technical architects, decision-makers, etc.) have adopted microservices and 92% are experiencing "at least some success" with microservices.

Microservices may be *polyglot*, meaning that they can be composed of services written in various languages such as Python and Java. Microservices developers using Java could leverage Jakarta EE (previously known as Java EE) which is an an extension of Java for building enterprise web applications [76]. Jakarta EE provides features specifically for development of web applications, such as Jakarta RESTful Web Services (JAX-RS) for creating RESTful HTTP servers [27]. However, while Jakarta EE is a well-known and established standard for creating standalone (monolith) enterprise applications, there are Java frameworks that focus on building enterprise *microservices* applications such as MicroProfile.

MicroProfile is a collection of specifications that describe a gap-filling functionality for development of microservices [43]. MicroProfile is primarily leveraged through Java annotations. In other words, client developers use the functionality provided by MicroProfile through annotation-based Application Programming Interface (API). MicroProfile heavily resorts to annotations as its API for client developers for several reasons: first, MicroProfile extends

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
@InterceptorBinding
@Inherited
public @interface Asynchronous {

}
```

Listing 1.1: Declaration of the `@Asynchronous` annotation in MicroProfile [17].

Jakarta EE and wants to stay consistent because it is often used alongside, as well as is build on top of, Jakarta EE [30]. Further, MicroProfile is a collection of abstract specifications and not a concrete server runtime, i.e., server runtimes that are MicroProfile-compliant, such as WildFly and Open Liberty, provide their own implementation for the required behavior outlined in MicroProfile specifications.

To support easier development of microservices in Java enterprise applications, it would be useful to have automated tools that can check for correct annotation usage. Checking for correct annotation usage is not specific only to MicroProfile or other microservices frameworks. In fact, it has been a problem since the introduction of annotations in Java in 2004 [10]: the built-in language semantics used for declaring annotations (e.g., `@Target` in Listing 1.1 indicates that an annotation can be applied on a method or class level) are not expressive enough to encode more complex usage rules. For example, the rule stating that "if you apply annotation `@Asynchronous` on a method, then the method must have a return type `Future` or `CompletionStage`" cannot be encoded in annotation declaration of `@Asynchronous` due to language semantics limitations.

There has been lots of efforts designing specification languages to encode annotation usage rules and checkers to ensure these rules are applied [31], [36], [42]. However, with the exception of one tool that focuses on mining XML co-configurations [86], most of these annotation checkers assume that someone will write the rules or that these rules are documented in the documentation.

Unfortunately, such rules are not always documented and expecting framework developers to exert extra effort to encode them in some specification language is not practical.

Ideally, such rules should be automatically inferred or at least a starting point is provided to framework developers for easy modification into a specification that can be then automatically checked by static analysis checkers to detect and prevent usage violations.

The idea of automatically mining general API usage rules, not specific to annotations, has been extensively researched in the literature [3], [4], [57], [70]. A common underlying premise is that frequent usages indicate rules that should be respected. However, most of these efforts do not take metadata (i.e., annotations) into account and focus on verifying API usages with control and data flow relationships, which are not necessarily applicable to annotations where order does not matter. While order does not matter, there might be complex association relationships such as ones between annotations and configuration files. In this work, we investigate whether the same idea of pattern based discovery of rules can be applied to mining annotation usage rules.

1.1 Research Questions

In this thesis, we aim to answer the following two research questions:

RQ1: How effective is pattern mining in discovering annotation rules? Our goal is to investigate whether frequent itemset mining, a simple and popular pattern mining technique, along with generation of candidate rules can mine actual annotation usage rules. We analyze how much the generated rules have to be edited (number of edit operations) to become actual usage rules. We focus on API usage rules of MicroProfile based on our industry partner's needs. To show generalizability of our approach, we additionally apply our approach on Spring Boot, a similar annotation-based framework for building microservices applications.

RQ2: How common are violations of the mined annotation rules? To show usefulness of the mined rules, we encode the mined and confirmed candidate rules static checkers and scan for violations in client projects that use MicroProfile and Spring. We also analyze Stack Overflow posts for manifestations of violations of the mined candidate rules.

1.2 Study Overview

To answer the above-mentioned research questions, we first analyze various documentation sources and Stack Overflow threads to find examples of annotation usage rules and, if applicable, places in code where these rules are violated. Based on the examples we found, we identify 5 categories of annotation misuse problems in MicroProfile (we focus only on 3 categories in this thesis) and infer a set of 8 code facts that we need to track to mine the rules.

We then use frequent itemset mining to mine frequent patterns from 493 MicroProfile client projects. To increase our chances of mining meaningful rules, we develop a set of heuristics to filter out meaningless patterns. To validate if this technique can indeed mine useful rules, we present the mined rules to our collaborators from IBM, one of whom is a contributor to MicroProfile project. We ask them to confirm, deny, or suggest edits to the mined candidate rules. Out of 36 potentially correct candidate rules, we extract 14 unique actual usage rules. These results show that pattern mining is feasible for extracting usage rules, though the mined rules require some editing (about 4 edit operations) to become fully correct.

To determine whether our approach to mining annotation rules is generalizable, we additionally applied it to Spring Boot, a similar Java framework that can be used to build microservices and that is also primarily used through annotations. We fetched 281 projects from GitHub and mined 17 semantically unique rules that we confirmed with documentation and team members who are expert users of Spring Boot. These results show that our approach is generalizable to annotation-based enterprise microservices frameworks that are similar to MicroProfile.

We then encode the mined usage rules for MicroProfile and Spring Boot to scan for violations in client projects. For MicroProfile, we scan the commit history for potential violations in the same set of projects that we use for mining. For Spring Boot, we fetch more than >1K projects which we use for scanning (at the latest commit). We find 12 violations in the commit history of 9 projects that use MicroProfile, as well as 28 violations in 24 projects that use Spring Boot API. For Spring Boot, we manually validate these violations and then report issues in the corresponding repositories. Overall, these results show that while the mined annotation usage rules are commonly violated by client developers and have a real-world impact, they are detected and fixed early in the development process.

1.3 Thesis Contributions

In this thesis, we make the following contributions:

- Infer types of code elements involved in annotation usage rules for enterprise microservices frameworks and categorize the usage rules by these types.
- Devise a simple and feasible first-of-a-kind approach for mining annotation usage rules for enterprise microservices framework/library APIs.
- Develop heuristics that increase the diversity of usages and decrease redundancy of the mined rules.
- Provide and discuss results of our pattern mining approach, which include automatically extracted rules that are confirmed with domain experts.
- Discover violations of the automatically extracted usage rules in client projects of two popular Java frameworks for developing microservices applications.

Overall, our results show that mining annotation-based rules is feasible, but has limitations such as presence of unnecessary information or lack of the necessary one of the generated rules. On one hand, unnecessary information comes from one of the disadvantages of frequent itemset mining, the pattern mining algorithm that we use: it is prone to generating unnecessary information that needs to be removed somehow, whether manually by hand or using some heuristics [23]. On the other hand, the lack of necessary information within the mined rules comes from limitations of our representation, i.e., we do not encode all possible relationships due to their complexity. Mining complex rules requires either more sophisticated analysis, such as data or control flow analysis, or over-engineering our technique towards the precise (but not generally applicable) relationships exhibited in usage rules of only one particular target API, thereby trading off generalizability.

1.4 Thesis Organization

The thesis is organized as follows. We provide necessary background and terminology in Chapter 2. After that, we review the related work in Chapter 3. We then introduce our approach in Chapter 4. Chapters 5 and 6 describe the evaluation setup and results. We discuss threats to validity of the results presented in this thesis in Chapter 7. We then discuss the results in Chapter 8 and conclude the thesis in Chapter 9.

Chapter 2

Background and Terminology

In this chapter, we provide necessary background information related to the topics of this thesis. We first introduce the microservices architectural style with a toy microservices application. We then introduce our target library, MicroProfile, which is a set of specifications that aims to facilitate the use of the microservices style for developing Java Enterprise applications. Finally, we introduce the concept of Java annotations, which is a form of metadata. We also provide several examples of MicroProfile annotations, which are a major part of MicroProfile API usages.

2.1 Microservices

Microservices refer to the *microservice architectural style* which is an approach to developing an application as a set of smaller ("micro") services, each running in a separate process [22], [79]. Traditionally, software developers have built web applications in the *monolith* style which constitutes an all-in-one, self-contained web application that includes user interface, business logic, and data access (see Figure 2.1). Modifying the monolithic application means rebuilding and redeploying the entire application. In addition, developers cannot just scale one specific function of the application, they have to scale the application as a whole.

Microservices solve some of the challenges monolithic applications have. They help build web applications as a suite of smaller (micro) services, each residing in its own process, thereby providing indepedent scalability and flexi-

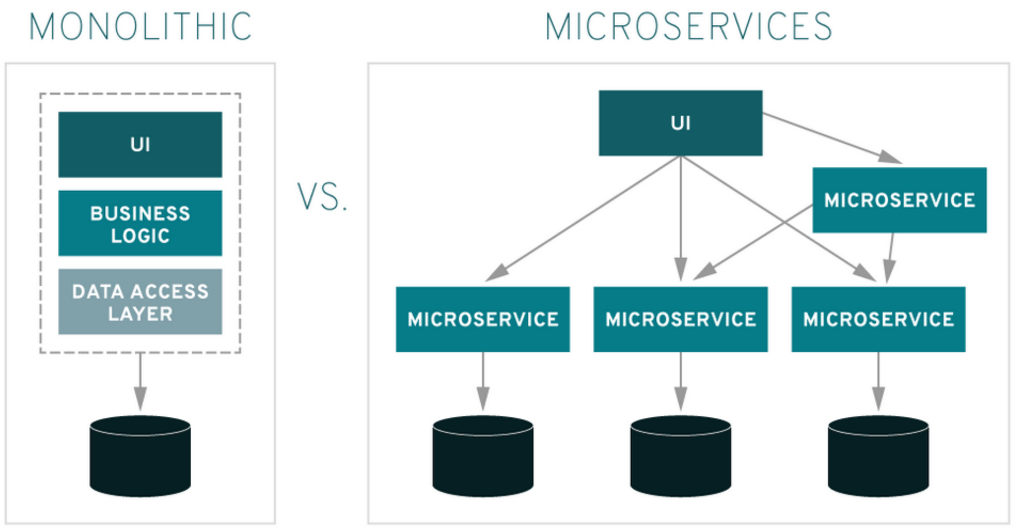


Figure 2.1: Monolithic vs microservices architecture [6].

bility. For example, if developers add or modify a feature, they do not have to redeploy the entire application – they can just rebuild and deploy the (micro) service that contains the updated feature. In addition, since services run in separate processes, they can be written in different programming languages (Java, Python, JavaScript, etc.) and may use different types of data storage (SQL, NoSQL, etc.). Many companies are pioneering the microservices style, such as Amazon, Netflix, Uber, and Groupon [52].

To better understand how microservices operate, let us go through the following scenario with Alice. Alice is a software developer who aims to build an online shopping store in the microservices architectural style in Java (as shown in Figure 2.2). She knows that there are customers that want to scroll through sold items, add some into the cart, and finally check out and pay for the products. For the check out functionality, Alice needs an *online cart* service that will be showing the users the products they are about to buy, as well as processing users' payments. She also wants to track what customer is buying what products, and therefore Alice needs a microservice that is responsible for *user authentication*. There needs to be another service such as the *products* microservice that will be responsible for processing (adding, retrieving, and modifying) products sold on the website.

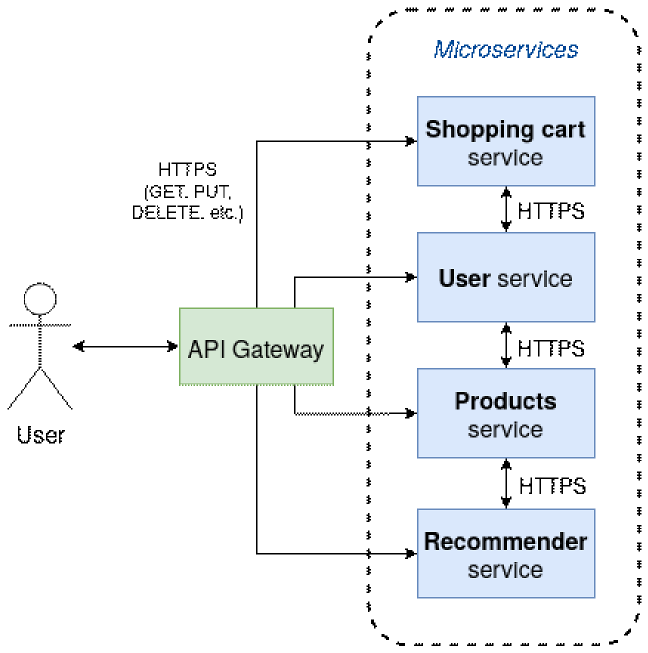


Figure 2.2: Alice's online shopping store system.

When users access Alice's store, they will access the *API gateway*. The API gateway is a single entry point for all users, and it interacts with some or all back-end services as needed. Since the back-end services need to receive and send data back to the gateway, Alice embeds a separate RESTful API into each service. One can also assume that each service comes with its own, separate database.

Some microservices might depend on others and thus, need a way to communicate with each other. Alice can choose a communication protocol depending on her needs, such as HTTP, TCP, and AMQP. She decides to stick with HTTPS for all her client-to-gateway and service-to-service communications.

Let us now consider the problems that Alice might experience as her system grows larger. With the growing number of features on her website, the complexity of the system grows along the way. With higher complexity, it becomes tedious for her to deploy, configure, and monitor each service individually. For example, there is a risk that some of the services might fail due to either connection issues or underlying hardware problems. The microservices that depend on each other need to be resilient when one of the services fails, i.e., she should not need to manually restart the service if it fails due to connection error. Alice realizes that she needs to make her system more (1) *fault-tolerant*, so that the users are still able to see and scroll through the items, even though they might not be able to immediately buy them due to the outage in one of the services. Besides fault tolerance, she needs to (2) *monitor* the performance of her microservices in order to avoid performance bottlenecks in her system that might slow down her website. Therefore, she needs some additional support for her microservices so that they are (1) more fault-tolerant and (2) can be easily monitored by her.

MicroProfile could help Alice add extra support for the microservices in the system. To accomplish requirement (1), she could use MicroProfile Open Tracing API that allows her to add instrumentation to her application. Alice will be able to see how much computational and memory resources each service uses. To be able to monitor (observe) her services in terms of their performance, Alice might resort to MicroProfile Health and Fault Tolerance APIs. The Health API allows her to monitor health of the services (whether the services are alive and ready to accept and process requests), and the Fault Tolerance API allows her to specify fallback mechanisms in case of the connection failures.

Figure 2.2 shows the overview of Alice's system. Having seen how some of the MicroProfile components could help Alice with engineering the microservices, let us now explore MicroProfile in detail.

2.2 Java Frameworks for Building Microservices

In this work, we focus on two Java frameworks for building microservices, namely MicroProfile and Spring Boot.

MicroProfile is an open-source Eclipse project aimed that optimizing Enterprise Java for building microservices [43]. It is a collection of APIs aimed to facilitate development of microservices in Enterprise Java web applications. It contains specifications for Open Tracing, Health, Fault Tolerance, Metrics

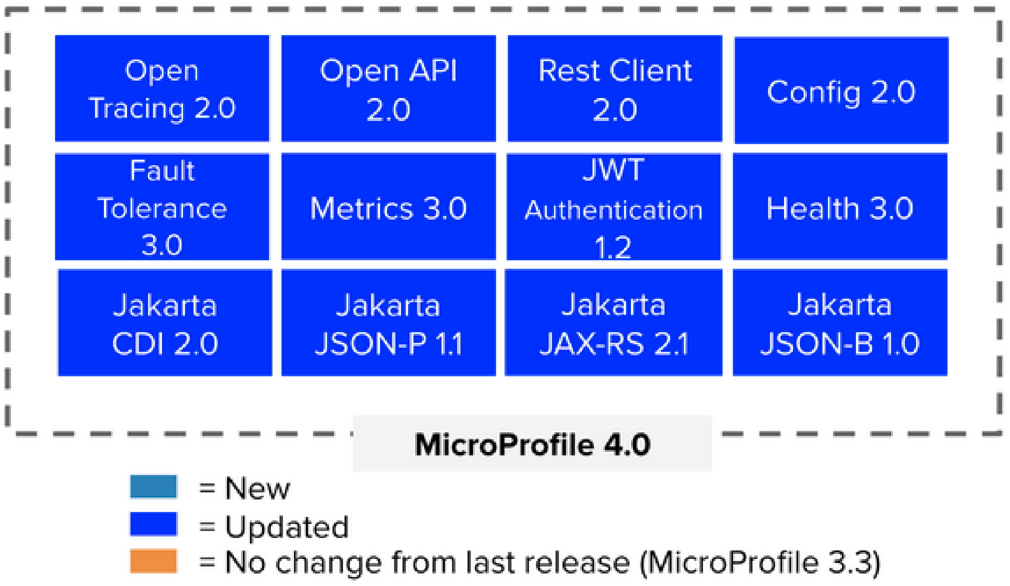


Figure 2.3: MicroProfile APIs/components [51].

and others.

MicroProfile builds on top of Jakarta EE (previously referred to as Java EE), which is a collection of standards for developing enterprise applications [76]. There are numerous web application projects that are Jakarta-EE compatible, such as Apache TomEE, Eclipse GlassFish, Open Liberty, WildFly, and JBoss Enterprise Application Platform [7], [21], [69], [74], [75], [88]. Unlike Jakarta EE which is a set of standards for building monolith enterprise applications, Microprofile is a set of specifications that aim to bring microservices to the Enterprise Java community.

Similar to Jakarta EE, using MicroProfile alone will not produce any result because it is simply a set of abstract specifications. To properly use MicroProfile, developers also need to configure a MicroProfile-supported runtime. There are multiple server runtimes that implement MicroProfile specifications, such as TomEE, WildFly, Quarkus, and OpenLiberty.

To better address the engineering needs of microservices, MicroProfile provides specifications as separate components (see Figure 2.3). For example, the Open Tracing component is used for distributed tracing and monitoring of microservices. If developers want to describe their APIs, they can use the

```java
import org.eclipse.microprofile.faulttolerance.Asynchronous;
import java.util.concurrent.CompletionStage;
import java.lang.String;

public class Foo {
    // Correct usage of MicroProfile Fault Tolerance API
    @Asynchronous
    public CompletionStage<String> doSmthAsync() {
        ...
    }

    // Incorrect usage
    @Asynchronous
    public String thisIsWrong() {
        // Oops! A runtime exception is thrown
    }
}
```

Listing 2.1: Example of a method-level annotation in MicroProfile.

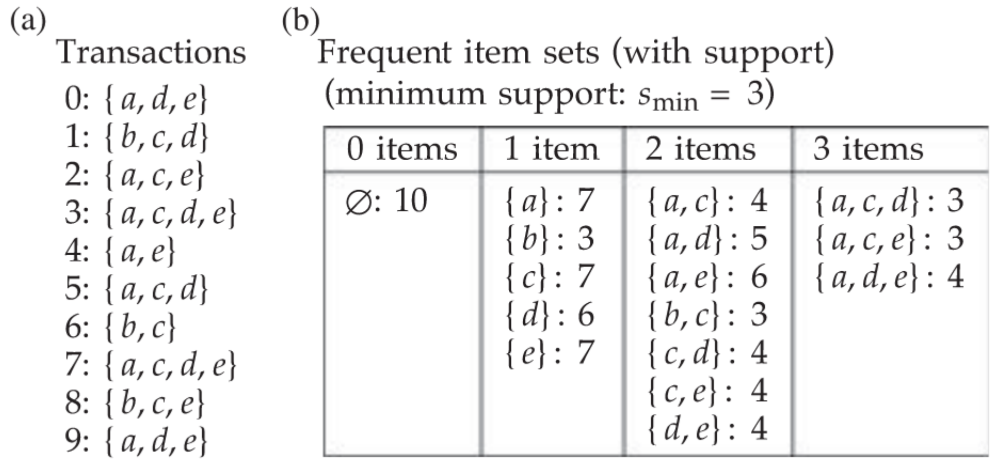
Open API component. In addition, to monitor health status of the services, as well as provide fallback mechanisms, developers may resort to the Health and Fault Tolerance components. For description of other components, see the official website for MicroProfile [43].

Java annotations. Throughout the components, MicroProfile provides the functionality primarily through *Java annotations*. Java annotations are a form of metadata applied on a variety of language constructs, such as classes, methods, method parameter, and constructors [60]. Annotations provide a convenient way of applying additional behavior to the constructs, whether the developer wants them applied at compile or run time. A major advantage of using annotations is reduced boilerplate code, i.e., pieces of code that have to be repeated in multiple locations with little to no change [32].

Consider Listing 2.1 that contains one of the annotations from the MicroProfile Fault Tolerance API, namely the annotation `@Asynchronous`. This annotation is both class and method-level, meaning that it can be applied on a class definition or method signature (as indicated by `ElementType.METHOD` and `ElementType.TYPE` values of the `@Target` annotation on the declaration of `@Asynchronous` in Listing 1.1). If `@Asynchronous` is used on a method, then the method will be invoked in a separate thread, as shown in Listing 2.1. If a class is annotated with `@Asynchronous`, then all the methods of that class will be invoked in a separate thread.

Even though using the annotation seems very simple, it comes with an additional constraint. As shown in Listing 2.1, if a method is annotated with `@Asynchronous`, then it must have a return type `Future` or `Completion-Stage<T>`. Otherwise, a runtime exception `FaultToleranceDefinitionEx-ception` occurs. This is an example of annotation usage rule in MicroProfile explicitly described in documentation [15]. Even though it is caught as soon as the application server starts up, large-scale applications with numerous dependencies take time to compile. Developers might not notice the mistake and waste time trying to build and deploy the application, whereas such violation could have been detected as soon as they typed the code if the usage rule could be explicitly encoded and statically validated in real time by some IDE.

Spring Framework (including Spring Boot) is a "world's most popular" web application framework for the Java platform [82], [83]. Like MicroProfile, Spring Framework offers a range of functionality, such as web app configuration, security, and GraphQL support, that is often used through Java annotations. The framework is a backbone of *Spring Boot*, which is an extension of Spring Framework that makes it easier to create stand-alone, production-grade Spring applications. According to the official Spring Framework website, "Spring Boot's many purpose-built features make it easy to build and run your microservices in production at scale". Spring Boot helps client developers create standalone applications by embedding a web server (e.g., Tomcat) into a client application. In addition to embedding a web server, Spring Boot provides *autoconfiguration* feature, meaning that an application comes with pre-defined dependencies that client developers do not need to configure manually. Such feature automatically configures the underlying Spring Framework and its modules (e.g., Spring Security) based on best practices. Thus, Spring Boot is simply an extension of Spring Framework that allows client developers



(a)

Transactions

0: $\{a, d, e\}$
1: $\{b, c, d\}$
2: $\{a, c, e\}$
3: $\{a, c, d, e\}$
4: $\{a, e\}$
5: $\{a, c, d\}$
6: $\{b, c\}$
7: $\{a, c, d, e\}$
8: $\{b, c, e\}$
9: $\{a, d, e\}$

(b)

Frequent item sets (with support)
(minimum support: $s_{\min} = 3$)

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

Figure 2.4: (a) An example of a database of 10 transactions (transactions from all projects) and (b) the frequent itemsets in it where the minimum support is at least 3 transactions [9].

to quickly create standalone microservice applications in Spring without much configuration.

2.3 Pattern Mining

Frequent itemsets. To extract annotation usage specifications as rules from a large amount of client code, we leverage a well-known pattern mining algorithm known as frequent itemset mining. *Frequent itemset mining* is a task of extracting patterns from a database of "transactions" [9]. It is primarily used for *association rule learning* [62], a machine learning technique for discovering interesting relationships between items in large databases. To discover patterns within some database, it is common to mine frequent itemsets and then use them to generate association rules.

Consider the following formal definition of frequent itemset mining [9]. Let $T = \{i_1, i_2, ..., i_n\}$ be a set of items, called *input itemset*, and $D = \{T_1, T_2, ..., T_m\}$ be a set of all input itemsets, called *database*. Alternatively, one can think of input itemsets as transactions that contain a list of items, such as bread and jam. Note that we refer to input itemsets as transactions in order to avoid confusion in terminology.

The term *itemset* refers to any subset of transaction T. For example, for transaction $T = \{a, b, c\}$ of size 3, there are $2^3 = 8$ possible itemsets (equivalent to power set of a set), namely $\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}..., \{a, b, c\}\}$. Each itemset I comes with the notion of *support* $supp(I)$, which calculates the frequency of transactions I appears in. In other words, if $supp(I) = k$, then I appears in k transactions. Given user-provided *support threshold* $supp_{min} \in \mathbb{N}$, itemset I is called *frequent*, if and only if $supp(I) \geq supp_{min}$. In other words, a frequent itemset is an itemset with support that is greater than or equal to user-specified support threshold.

Figure 2.4 shows an example of a database of 10 transactions (equivalently, 10 input itemsets created from all projects used for mining). Given minimum support threshold $supp_{min} = 3$, we get 16 frequent itemsets with support values shown in the table on the right. The frequent itemsets' sizes range from 0 (empty frequent itemset) to 3 items (e.g., $\{a, c, d\}$).

To retrieve frequent itemsets, we leverage the parallelized version of a popular mining algorithm called *FP-Growth* [20], [34]. Unlike traditional *apriori mining* that generates numerous candidate sets and then filters out non-frequent ones [2], FP-Growth leverages a tree structure, called *FP-Tree*, to mine a set of frequent itemsets. FP-Growth is more efficient because, unlike apriori mining which scans database over and over, FP-Growth scans the database only two times. For more detailed information on FP-Growth, see [20].

Association rules. Once frequent itemsets are retrieved, they can be used to generate candidate association rules. *Association rules* (or candidate rules) are the relational rules of the form "If X, then Y" or $X \implies Y$ where I is a frequent itemset and $X, Y \subseteq I$. 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 $X \implies Y$ is a measure of how often a candidate rule $X \implies Y$ has been found to be true:

$$conf(X \Longrightarrow Y) = supp(X \cup Y)/supp(X)$$

where X, Y are frequent itemsets. For example, if $X = \{$ *"jam", "peanut butter"*$\}$ and $Y = \{$ *"bread"*$\}$ and $conf(X \Rightarrow Y) = 50\%$, then it means that 50% of transactions that have *"jam"* and *"peanut butter"* also have *"bread"*.

While the minimum support threshold $supp_{min}$ helps us mine frequent itemsets within a database of transactions (or input itemsets), the user-defined minimum confidence threshold $conf_{min}$ helps us generate candidate (association) rules with strong "if-then" relations between items. In other words, we generate an association rule $X \Longrightarrow Y$, if and only if $conf(X \Longrightarrow Y) \geq conf_{min}$.

In this thesis, we use frequent itemsets to generate candidate rules that potentially represent usage specifications. We present our pattern mining approach in Chapter 4.

Chapter 3

Related Work

In this chapter, we discuss existing work related to the topic of this thesis, i.e., mining annotation-based usage rules.

An *Application Programming Interface* (API) is an interface through which a software library or framework is used. When a developer wants to bring in some functionality to their software, they resort to reusing existing libraries or framework through APIs. The purpose of an API is to abstract away the implementation details, so that the client developers reuse the functionality they need without necessarily knowing how it is implemented [26].

When using APIs, developers often make mistakes, known as *API misuses*, that result in bugs and even security vulnerabilities [4], [37], [53], [56], [78], [85], [87], [90]. These violations could produce unexpected behavior such as program crashes and resource leaks. One way to prevent API usage violations is to write *API usage specifications* that precisely define how to use an API, such as by outlining the necessary steps and dependencies. Our goal is to automatically mine such specifications. Accordingly, we divide related work into 2 categories, manual writing of API usage specifications and automated mining of API usage specifications, which we discuss below.

3.1 Writing API Usage Specifications

General usage specification DSLs. To prevent API misuses, one can create automated static checkers that check client code for correct usage of some API. However, the automated checkers need usage specifications that describe correct usage of some API. One can write usage rules as specifications in some domain specific language. In contrary to general programming languages such as Java, a *domain-specific language* (DSL) is a language used only for specific domain [19]. For example, CrySL is a DSL in which cryptography experts can write cryptography API usage rules [31]. Researchers in the CrySL study manually encoded the usage rules by first producing a set of rules based on documentation and then refining these rules through discussions with cryptography API experts. The written specifications are then compiled into static analysis checkers that automatically scan for API usage violations. Other examples of DSLs and similar tools include uContracts [41] and .NET Code Contracts [18]. uContracts [41] is a DSL for specifying "usage contracts" between two parties: a provider (e.g., an API) and a consumer (e.g., client code using the API). According to Lozano et al, a usage contract "defines the expectations and the assumptions by the reusable entities [of the API] on the entities that reuse it [client code]". While CrySL and uContracts are external and separate languages, the code contracts in .NET Framework are part of the platform itself, i.e., all .NET Framework languages (e.g., C#) can take advantage of the contracts without any external tooling support. They allow client developers to specify additional constraints on their client code using pre-conditions, post-conditions, and invariants using .NET Framework `System.Diagnostics.Contracts` package. For example, a precondition `Contract.Requires(x != null)` expresses that parameter `x` must be non-null. The contracts come with a built-in static checker that verifies the usage constraints without running the program. However, both .NET code contracts and uContracts do not provide support for specifying and verifying annotations or attributes[1].

Annotation-based usage specification DSLs. There are also DSLs designated for writing annotation-based usage specifications. Darwin proposes a DSL, called AnnaBot, for verifying annotation usages [11]. Similar to the nature of our candidate rules (i.e., rules are of the "if-then" form), their DSL

[1]An equivalent of a Java annotation in C# and other .NET Framework languages.

uses implications for writing rules ("If X, then Y"); provides *checks* that check for existence of an annotation on the class, method, or field; as well as logical operations such as AND, OR, and NOT for specifying relations between annotations, types, etc. Similar to Annabot, Eichberg et al. [16] present a user-extensible tool for automated annotation-based checking of implementation (usage) restrictions. They write the restrictions as checkers in XQuery, a query language for XML documents. Given some client Java code, they convert it into XML and run the checkers (pre-defined queries) to validate the usage constraints in the code. A work done by Kellens et al. [29] provides custom annotations and logical predicates that add extra semantics on top of annotation declarations. Some of their logical predicates, such as *"?meth returns: ?exp"* for specifying method return type, are similar to the relationships or code facts that we track in code, such as "Method hasReturnType Foo". A tool developed by Laàzaro de Siqueira Jr. et al. [71] uses a DSL for validating annotation usage rules that may span different locations in code (e.g., annotations applied on top of other annotations' declarations). If one does not account for the possibility that annotation and its target element can be decoupled (i.e., an annotated program element such as class or method), there might be numerous false positives (false violations). Accounting for such annotation usage complexity is important for static checkers, but in our work, we focus only on evaluating feasibility of pattern mining for extracting annotation usage rules, as well as checking the usefulness of these rules in practice. In other words, highly accurate/precise static annotation usage checker is beyond the scope of this thesis.

In terms of mining annotation usage rules specifically for enterprise microservices frameworks or libraries, the thesis work by done Yaxuan Zhang is perhaps the closest related work [91] because they have a DSL that simultaneously supports 3 code entities we mine in our rules (discussed in Chapter 4.1.2). Zhang does not mine any rules, but rather offers a DSL that supports specifying relationships between annotations and configuration files. In other words, their DSL engine is capable of reading both Java and XML files for verifying the usage rules that involve both locations. Similar to Annabot [11] and other

DSLs, the DSL has loop constructs, such as the “for” loop, as well as existence assertions, such as “exists”. Their DSL is more expressive than the ones we discussed previously and allows for encoding of complex relationships, but it requires users to know the language in advance to encode usage rules which might get cumbersome.

Semi-manual approach to writing API usage specifications. Despite the numerous general and annotation-usage-specific DSLs, the DSLs require manual encoding of the rules, which is a tedious and error-prone process. To alleviate some of the manual work of writing specifications, Mehrpour et al. [42] offer a combination of a DSL and user interface (UI) called *RulePad*. RulePad provides snippet-based and semi-natural-language modes of authoring rules. Although our work does not incorporate manual writing of API usage rules, we ask domain experts to confirm, deny, or suggest edits to the candidate rules that we mine (see Chapter 5.1). We could potentially leverage RulePad to automate the process of verifying mined usage rules because the DSL supports annotations and the relevant relationships (see Chapter 4.1.3). Even though RulePad makes the usage specification writing process easier, it might be still tedious and time-consuming for developers to encode all the usage specifications for APIs that have large usage surface (with numerous public methods, classes, types, etc.). Ideally, we want to infer API usage rules using automated techniques or at least provide developers with a starting point for easy modification into specifications.

3.2 Mining API Usage Specifications

Writing API usage specifications manually is difficult and time-consuming. Therefore, researchers have proposed to use data (pattern) mining techniques to extract API usage specifications automatically [54].

The general assumption behind such work is that when dealing with massive amounts of client code that uses some target API, majority of *repeated* usages are correct. In other words, frequently occurring usages often imply correct usages. One might then collect a massive amount of client code and apply data mining methods to find *usage patterns*, parts of usages that commonly occur within all usages. Researchers often use these patterns to verify whether some API usage is correct or incorrect by looking for deviations of some given API usage example from the patterns [4].

Mining general usage patterns from code artifacts. According to Robillard et al., an *API property* is "any objectively verifiable fact about an API or its use in practice" [70]. There are different approaches to automatically infer these "facts": non-sequential, sequential, behavioral, migration mappings, and general information [5], [28], [38]–[40], [54], [65], [66]. For example, Acharya et al. collect static program traces of API elements of interest (e.g., public methods) and use these traces to mine frequent partial orders, which are then converted into usage specifications [1]. Zhong et al. also analyze client code, but track API call sequences contrary to partial orders [92]. In our work, we utilize a non-sequential approach because there is no order relation among annotations. We resort to simple frequent itemset mining that mines frequent associations between annotations and other relevant elements. Similar to our approach, Li et al. leverage a non-sequential (unordered) frequent itemset mining algorithm to extract associations among program elements, such as functions and variables (e.g., functions and variables) [35]. Even though the flow of their technique (how they extract patterns, generate rules, find violations, etc.) is almost identical to ours, they do not mine associations between annotations or between annotations and program artifacts (e.g., methods).

There are also more complex representations and techniques that can capture more complex usages (e.g., data and control flow) such as frequent subgraph mining. For example, Amann et al. present MuDetect [4], a static API misuse detector, based on previous work by Nguyen et al. [56]. In MuDetect, Amann et al. represent code as, and mine patterns from, API Usage Graphs. An *API Usage Graph* (AUG) is a graph-based representation that captures data and control flow properties within method bodies. Nodes represent variables, method calls, etc. and edges represent data and control flow relationships. Graphs can also be used to identify boilerplate code associated with APIs [55]. Although their graph mining approach can retrieve more complex patterns, the technique might result in memory overhead for large amount of client projects. In addition to potentially slow performance, using graphs for annotation-based API usages is excessive because annotations do not have complex data and order flow semantics, i.e., it only matters whether you apply certain annotation or not. Unlike MuDetect, we do not analyze method bodies and instead focus on mining usages that primarily involve annotations, program elements such as classes, fields, and methods (we track information only from their signatures and declarations), as well as configuration files.

Mining configuration usage patterns non-code artifacts. Our mining approach is able to mine rules that involve non-code artifacts, such as configuration files that are necessary for configuring microservices application servers (e.g., specifying a port for an HTTP server). Apart from mining usage patterns from code alone, there are other approaches that mine usages from related code artifacts, such as configuration files [86] and code comments [8], [73]. The mining approaches here are not based on *pattern* mining, but rather on regular expressions or similar parsing heuristics (e.g., looking for pre-defined words in text such as code comments). For example, Wen et al. mine configuration couplings (pairs of XML elements that frequently co-occur together) in *deployment descriptors*, which are the XML files used to configure applications [86]. While their work focuses on mining rules within these configuration (XML) files alone, we mine relations between annotations, program elements, and configuration files. On the other hand, when we analyze configuration files, we only look for presence of class names mentioned in code because our analysis looks for relationships between code and configuration elements, rather than between configuration elements only.

Mining annotation usage patterns. To the best of our knowledge, while there is a multitude of tools that leverage pattern mining techniques to mine usage patterns from code [53], [78], [84], [85], none of them mine annotation-based API usage rules that may additionally involve non-code artifacts, such as configuration files.

Chapter 4

Mining Annotation-based API Usages

In this chapter, we present the methods we use to mine annotation usage rules of enterprise microservices frameworks or libraries. Our goal is to use pattern mining to find annotation usage patterns that potentially represent usage rules.

An overview of our approach can be seen in Figure 4.1 where each rectangle is a step in our methodology. As the first step, we identify what kind of relationships (code facts) we need to track to mine usage patterns for annotation-based frameworks. After we parse code into itemsets of these code facts, we pre-process the itemsets to ensure that we mine as many diverse API usage patterns as possible. After we apply the frequent itemset mining algorithm to mine usage patterns as frequent itemsets, we post-process the frequent itemsets to get rid of invalid patterns and generate candidate rules in an "if-then" form.

4.1 Representation: Tracking Code Facts

APIs typically contain implicit usage constraints. These usage constraints are known as *usage rules* that API users should abide by. However, violations of these usage rules (API misuses) may lead to services crashing unexpectedly at runtime. Similarly, the same problems may happen when using annotation-based APIs, such as MicroProfile.

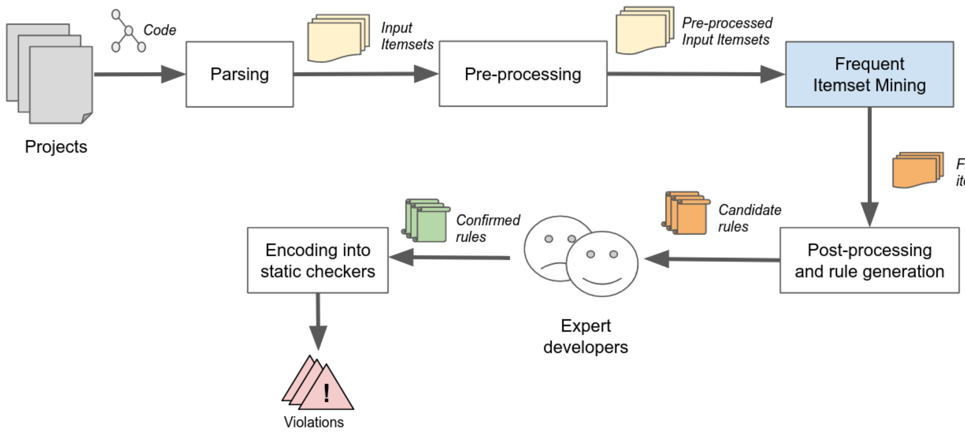


Figure 4.1: Overview of our mining and validation approach. Each rectangle represents a step/process in the approach.

4.1.1 Identifying Which Code Facts to Track

Annotation-based API usages are different from the API usages that existing pattern mining tools typically extract [1], [78], [92]. The existing work in the area of mining API usage patterns focus on usages within method bodies [4], [56], [84], [85]. As we discussed in Chapter 3, existing API usage extraction tools do not analyze annotation usages. Motivated by our industry partner's interest in MicroProfile usage, to understand how annotations are used and what needs to be tracked within code in order to be able to identify potential usage problems, we use MicroProfile API as a starting point, because it is primarily used through annotations. We search for explicit usage rules in an "if-then" form in the official MicroProfile documentation [43], as well as any issues client developers have faced when using MicroProfile API on Stack Overflow and official MicroProfile forum on Google Groups [46].

To identify existing issues with usage of MicroProfile annotations in Stack Overflow, we search for posts with tag *microprofile* [72]. As of June 23, 2021, there were 233 questions related to MicroProfile. We retrieve these questions using only `microprofile` tag as the search query. We look at each post title, body, comments, and all answers on that post. We ignore questions that ask for general information and do not contain any code snippets, such as "Why Eclipse Glassfish does not support Eclipse Microprofile" [81] and "WebSphere

Application Server support of MicroProfile" [25]. We also skip questions that have questions specific to some runtime only and not MicroProfile, such as "Wildfly 17 error "WFLYMETRICS0003: Unable to read attribute second-level-cache-hit-count" when statistics-enabled="true"" [33]. That question is concerned with a WildFly-specific exception and is not necessarily related to MicroProfile, and probably should not have the `microprofile` tag at all. While going through all the posts, we focus on questions where authors explicitly describe their question with code snippets as well as problems they experience (e.g., runtime exceptions). For example, in one of the questions, the question author asks about the issues they are experiencing with authorization in their application [68]. The problem is that the application deploys without any error and they do not get any logging information from the server, even though the authorization feature does not work as expected (i.e., should return HTTP 401 "Unauthorized", but returns 200 "OK" instead). Along with such questions, we look for concrete answers that follow specific text patterns, such as "If you are using X, you should/must/need/have to use Y", "X must/should/need/requires/misses/has to have Y", or "To do X, you must/should/need/have to have Y". These text patterns are similar to the candidate rules we want to generate. For example, for the same question above, the accepted answer is found in the following sentence:

> *Your JAX-RS configuration class is missing the* ***@DeclareRoles****({ "mysimplerole", "USER" })*

The answer above may be turned into a rule such as "If there is *@DeclareRoles* on any method of any of class, then the class that extends *Application* must be annotated with *@LoginConfig*" because declaring user roles on some HTTP resources is part of *user authorization* (i.e., managing user access level). Violating the rule above (i.e., using *@DeclareRoles* without *@LoginConfig* on the *Application* class) results in faulty behavior that has no visible error: the developer expects to see user authorization feature based on user roles on a given HTTP resource, whereas in reality it is simply not there.

In the official MicroProfile forum on Google Groups, we find only 1 question related to API usage of MicroProfile because the forum is designated to facilitate general conversations about MicroProfile and its future plans, rather than its usage. For example, some posts discuss new releases[44], [47], provide feedback on existing specifications [48], [50], and discuss future plans [45], [49]. Since the majority of the posts discuss only general information about MicroProfile, we do not find specific and technically detailed information about its API usages that could help us understand what can go wrong (e.g., what developers typically miss) when using MicroProfile annotations.

In total, we extract 15 usage rules, out of which 10 are from the official documentation, 4 are from Stack Overflow and 1 is from the forum on Google Groups. We additionally verify the 5 rules from Stack Overflow and Google Groups with our collaborators from IBM, some of who are direct contributors in the MicroProfile project, to confirm or deny whether a potential rule is a rule or just a common idiom. We do not need to verify the 10 rules from the official documentation because they are explicitly described there.

4.1.2 Categories of Usages

We examine the 15 usage rules we manually extracted above (we share them publicly on GitHub [59]). Our goal is to identify what code elements appear in these rules, as well as the relationships between these elements. Understanding this can help us determine the code facts we need to encode in the representation we use for pattern mining. We observe that the 15 usage rules we analyzed involve the following 3 types of code elements:

- **Annotations (ANN):** Annotations are the primary entities found in the usage rules we analyze. Unlike marker annotations that have no parameters, such as the built-in annotation `@Override`, MicroProfile annotations tend to have one or more parameters (e.g., `@Fallback(X.class)` where `X` is a locally defined class).

- **Configuration (CONF):** This primarily involves configuration files such as `beans.xml`, which contains bean class configurations, and `microprofile-`

`config.properties`, which contains key-value pairs specific to the web application. Both files are used to separate configuration from code.

- **Program elements (PE):** Program elements come from code itself and include elements such as constructors and method signatures. In the rules above, we observe the following program elements: method parameter list and return types, constructor parameter list types, field type, class extensions (including interface implementations), as well as method body contents, such as constructor and method calls.

The majority of rules we find span multiple entities, and thus, include elements of multiple forms such as annotations and configuration files. Since some rules are a mix of different forms, we classify the rules into 5 categories:

- **Annotation–annotation (ANN-ANN):** This category includes rules that express relationships only between annotations (e.g., usage of one annotation requires another). For example, if a developer applies annotation `@RolesAllowed` on any method of any class, then the developer should annotate the driver class (i.e., the one that extends `javax.ws.rs.core.Application`) with `@LoginConfig`. In other words, `@LoginConfig` enables the authorization feature that allows defining user roles (e.g., "admin" vs "user") for HTTP endpoints in the application [14].

- **Annotation–program element (ANN-PE):** This category contains rules that involve annotations and program elements (e.g., classes). For example, if a developer annotates a class with annotation `@Liveness`, then the developer should make sure that the same class implements interface `HealthCheck` [13].

- **Annotation–configuration (ANN-CONF):** This category contains rules that involve annotations and configuration files (e.g., `microprofile-config.properties`). For example, if a field is annotated with `@ConfigProperty(name="foo")`, where the sole annotation parameter `name` has value `foo`, then `foo` must be defined in `microprofile-config.properties` file or any other configuration source [12].

- **Configuration–configuration (CONF-CONF):** This category includes rules that involve configuration files only (e.g., `beans.xml` file). For example, the usage of the XML element `<ssl id="x" keyStoreRef="bar" />` requires that value of the attribute `keyStoreRef` (`"bar"`, in this case) to be defined in another XML element `keyStore` (in the same file) as the value of the attribute `id`, like `<keyStore id="cacertKeyStore" ... />` [89].

- **Program element–program element (PE-PE):** This category contains rules that involve program elements only (e.g., method bodies). For example, if a developer calls method `build()` of an object of class `RestClientBuilder`, then the developer has to extend related class `RestClientBuilderResolver` and override the method `newBuilder()` in the same codebase. The method `RestClientBuilderResolver.newBuilder()` is then going to register the custom builder and is going to return the builder that is going to execute the method `build()` that was called previously [64].

Notice that we do not list the combination **Program element–configuration (PE-CONF)** category where rules have relationships only between program elements (e.g., classes) and configuration files because we did not find any such rules for MicroProfile, based on the sources we analyzed, even though previous studies do mention such types of rules [86]. In addition, there is only one rule that has all three code elements (i.e., `ANN`, `PE`, and `CONF`).

We present the number of extracted rules grouped by category in Table 4.1.

Studying these collected rules and understanding which entities they involve allow us to determine which relationships and code facts we need to track to automatically mine these rules. In this work, we focus only on the first 3 categories of the rules.

4.1.3 Supported Relationships

We now discuss relationships (or code facts) that we track between different elements in a program. We consider **annotations**, **program elements**, and

Category	**# of rules extracted from documentation**	**# of rules extracted from Stack Overflow or other forums**	**Total # of rules**
ANN-ANN	2	1	3
ANN-PE	6	0	6
ANN-CONF	1	0	1
CONF-CONF	0	3	3
PE-PE	1	1	2
Total	**10**	**5**	**15**

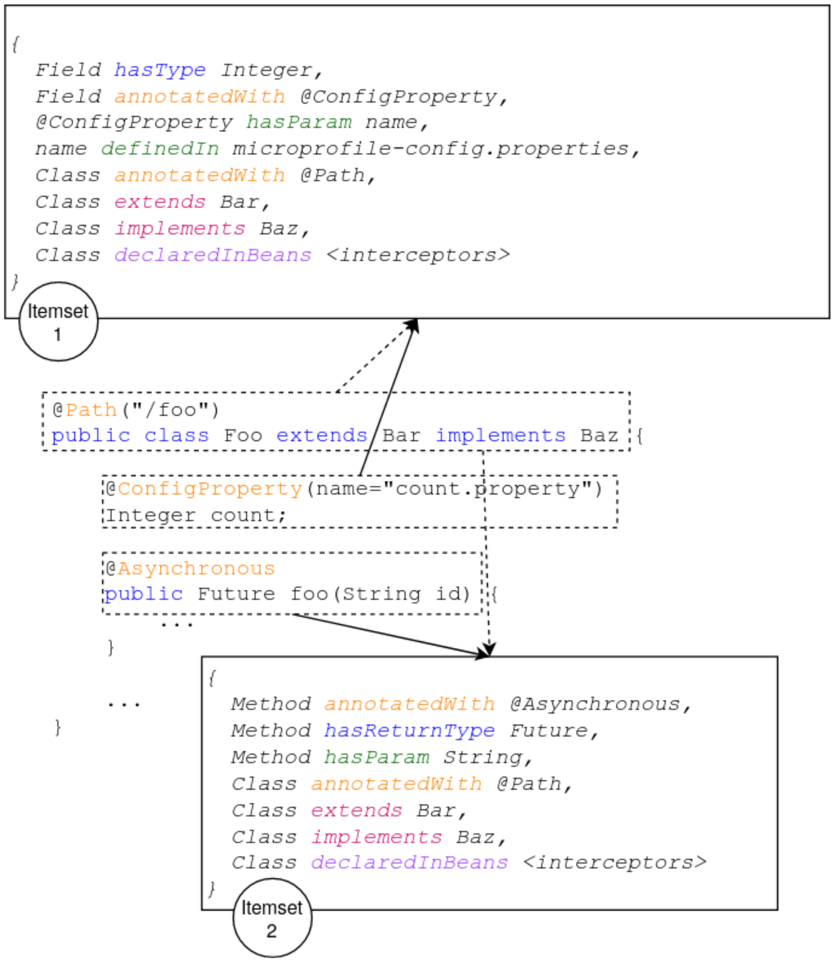
Table 4.1: Categories of manually-extracted rules.

configuration files as the entities in the relationships we track.

While the entities contain a plethora of detail within them, we selectively track only a subset of information based on observations from the manually extracted rules. With annotations, we track their parameters. With program elements, we track information only about fields, method signatures, class signatures, and constructor signatures, such as type and class hierarchy (if applicable). In terms of configuration files, we analyze the `microprofile-properties.config` file that stores pairs of keys and values for configuring the MicroProfile runtime; as well as `beans.xml` which contains class configurations that can be changed at load time.

Figure 4.2 depicts an example of a code snippet and configuration files, as well as the relationships that we support. We do not display fully-qualified type names (i.e., `org.a.b.c.ClassName`) for the sake of brevity, but we do use them for differentiating annotations and types that have the same name but come from different libraries and packages.

In Figure 4.2a, there are three entities that we keep track of: class, field, and method. We store information about fields and methods separately. In the figure, we have the "Itemset 1" box that contains an itemset that represents the field `count` and the "Itemset 2" box that contains an itemset that represents the method `foo`. While we formally introduce itemsets in the next section, think of itemsets as sets containing these relationships. Note that we do not create a separate itemset for the class `Foo` and instead put all relevant



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

```
@Path("/foo")
public class Foo extends Bar implements Baz {
    @ConfigProperty(name="count.property")
    Integer count;

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

    ...
}
```

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

(a) Supported relationships for an example code snippet

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <interceptors>
        <class>org.example.project.Foo</class>
    </interceptors>
</beans>
```

(b) Example content of `beans.xml` file.

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

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

Figure 4.2: Supported relationships

class information into itemsets that represent class members such as fields, methods, and constructors[1]. We decided to not create a separate itemset for classes to enable us to mine potential relationships between classes (and their annotations or extensions) and fields or methods (and their types or annotations) based on our observations with the manually analyzed rules. In other words, if we create a separate itemset for a class, then the relationships of class and its member elements become isolated from each other, thereby removing the ability to mine cross-element relationships (i.e., between the class and methods or fields). In addition, the underlying structure is a *set* and sets do not allow for any duplicate values: while two methods return the same type T, we will track a return type of only one of these methods thereby incorrectly representing information within a class. We provide more information about how we track class information in the next section.

Further, when parsing class signatures, we do not keep track of class names, because that limits the possibility of the pattern mining technique to discover anything. Class names are too specific for frequent itemset mining to extract patterns. In addition, the names are irrelevant in mining association rules, because there are typically no rules that force the usage of a particular class name or to follow a particular convention apart from what compilers already check for, i.e., the validity of the name.

More formally, we track the following relationships when parsing client code in Figure 4.2a:

- The *annotatedWith* relationship tracks an annotation on some program element. We consider the following program elements: class, field, method or parameter. For example, in the "Itemset 1" box, we add the relationship "*Field annotatedWith @ConfigProperty*" because the field is annotated with `@ConfigProperty`. We create one relationship per annotation.

- The *hasType* relationship tracks the type of a field. For example, in

[1]Note that since there is no constructor in the code snippet in the Figure 4.2a, there is no itemset for it

the “Itemset 1” box, we add the “Field hasType Integer” because the represented field is of type `Integer`. There cannot be more than one of “hasType” relationships because a field has only one type. If we are dealing with generic types, then we simply ignore the type parameter list (e.g., we keep only `Optional` from `Optional<Integer>`). That way, we are more likely to mine a usage rule that involves the generic type. In other words, the more general the information we track, the more likely we are to mine usage rules that involve the generic type so that it is not constrained by type parameters. Note that we track parameter types separately using the *hasParam* relationship.

- The *hasParam* relationship tracks a parameter of some program element along with its type. For example, in “Itemset 2” box, we add the relationship “*Method hasParam String*”, which means that method has a parameter of type `String`. Similarly, we track annotation parameters, such as the “*@ConfigProperty hasParam name*” relationship in the “Itemset 1” box. This relationship tracks one parameter at a time.

- The *hasReturnType* relationship tracks a method’s return type. For example, we have the “*Method hasReturnType Future*” relationship that tracks the method’s return type, i.e., `Future`. This relationship is similar to *hasType* relationship.

- The *extends* relationship tracks class extensions of a given class. This relationship is established only for locally defined or target library/framework classes. In other words, we do not track class extensions that belong to 3rd party libraries. For example, in the “Itemset 1” and “Itemset 2” boxes, we add the “*Class extends Bar*” relationship assuming that `Bar` is a locally defined class (i.e., in the same project) or is part of a target library/framework we are interested in.

- The *implements* relationship tracks a class’s implementations of interfaces, if any. This relationship is similar to the *extends* relationship except that it tracks interface implementations (e.g., the "Class implements Baz" relationship in the "Itemset 1" and "Itemset 2" boxes).

- The *definedIn* relationship connects some annotation parameter value to a key defined in `microprofile-config.properties` configuration file. For example, the "*name definedIn microprofile-config.properties*" relationship tracks the `count.property` variable defined in the properties file (Figure 4.2c). This relationship is MicroProfile-specific and might not work for other frameworks.

- The *declaredInBeans* relationship connects a class to `beans.xml` file that typically contains class configurations. The `beans.xml` file originates in Jakarta EE, but is also an essential part of configuring beans in MicroProfile applications. The connection is established only if the fully-qualified class name is present in the configuration file. For example, we analyze the *beans.xml* file (Figure 4.2b) and see that the fully-qualified name of class `Foo` is mentioned under the <`interceptors`> parent XML element. We therefore add the "*Class declaredInBeans <interceptors>*" relationship to the "Itemset 1" and "Itemset 2" boxes.

4.2 Usages as Input Itemsets

After we establish what kind of code information and code relationships we want to track, we now proceed with converting client code API usages to itemsets where each item is a relationship that tracks a code fact (we discuss items and itemsets in Chapter 2.3).

We establish one input itemset (transaction) per **method**, **field**, and **constructor** to avoid ambiguity in differentiating methods and the information about them (return type and parameter type list). We initially tried only having one itemset per class that combined code facts for its annotations, methods, method annotations, parameters, etc. However, itemset representation is too simple and does not differentiate between method annotations, parameters and return types, i.e., we do not know which method parameter belongs to what method. We therefore decided not to proceed with representing a class with one large itemset.

Despite not creating an itemset per class, we still have to track code facts on a class level. As methods, fields, and constructors reside within a class and the class itself might have annotations, there might be associations between class annotations and annotations on some of these constructs. In addition, there might be a rule that involves both a class extension and an annotation on the same class. To avoid missing rules that involve annotations on different scopes, we replicate and add code facts of the class (e.g., class annotations) into all itemsets used to represent its fields, methods, and constructors, thereby allowing the mining of rules that involve annotations on and within the class. For example, in Figure 4.2a, boxes 1 and 2 are input itemsets that represent the field `count` and the method `foo`, respectively. The black arrow and the dashed rectangles around the field and the method signatures depict what relationships are there in each entity. In addition, the two dashed arrows show the code facts on the class that we replicate and add into each of the two input itemsets. For example, the relationship *"Class declaredInBeans <interceptors>"* is added to both field (box 1) and method (box 2) input itemsets. Given that there are usually very few (less than 10) to no annotations on class signatures, our replication approach does not cause any performance issues.

4.3 Mining Frequent Itemsets

After we convert usages into input itemsets (or transactions), we proceed to mining frequent itemsets.

We introduce definitions of an *item*, *itemset*, and a *frequent itemset* in Chapter 2.3. In our context, an item refers to one relationship, e.g., *"Class annotatedWith @Path"*. Frequent itemsets are then sets of relationships that frequently co-occur together in the set of transactions from all client projects that use target API, such as MicroProfile. For example, here we have 2 frequent itemsets that represent two methods, respectively:

```
[
    {
        "Method annotatedWith @Asynchronous",
        "Method hasReturnType Future"
    },
    {
        "Method annotatedWith @Asynchronous",
        "Method hasReturnType CompletionStage"
    }
]
```

The two frequent itemsets above show that methods that are annotated with `@Asynchronous` also frequently have the return type of *Future* or *CompletionStage*. As shown with these two frequent itemsets, frequent itemsets are frequent associations between various code entities (e.g., annotations and types) that could potentially be turned into an actual usage rule.

To retrieve frequent itemsets, we leverage the parallelized version of a popular mining algorithm called *FP-Growth* [20], [34], introduced in Chapter 2.3. While using FP-Growth helps us cut down the mining time, we are still likely to get a large number of frequent itemsets that are not very useful or too similar to each other (i.e., one frequent itemset can be a superset of numerous others). In addition, some frequent itemsets may simply represent common usage idioms, but not usage rules. As we discussed earlier, there can be potentially 2^n frequent itemsets for a transaction of size n. To alleviate the redundancy issue and take only the most interesting frequent itemsets, we post-process the frequent itemsets, as discussed in the next section.

4.4 Post-processing Frequent Itemsets

Before we generate any rules from the retrieved frequent itemsets, we have to perform several post-processing steps to minimize as much redundant information as possible. These steps allow us to decrease the likelihood of generating "noise". The noise in case of frequent itemsets can take several forms: (1) a frequent itemset that solely contains usage that is a common idiom and not a usage specification; (2) a frequent itemset that contains elements that cannot be used in any way to generate a semantically correct candidate rule (i.e., there is no rule that accurately describes something and logically makes sense); (3) a frequent itemset that can be used to generate a candidate rule, but contains some irrelevant relationships that have to be later edited (removed or modified). Therefore, we aim to remove as many irrelevant frequent itemsets as possible before we generate candidate rules.

Based on our trial-and-error observations as well as discussions with the our collaborators from IBM, we develop a set of heuristics that help us bring the noise down and that are applicable irrespective of target library or framework. In the following list, we describe each heuristic one-by-one and provide motivation for each:

Focus on maximal frequent itemsets. To reduce the number of similar frequent itemsets, where one frequent itemset is a superset of another, we can leverage the *maximality* principle. In other words, we will only keep frequent itemsets that are *maximally frequent*, i.e., that have no proper frequent 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 $B = \{a, b, c\}$. A and B are very similar, B being a superset of A by inclusion ($B \supseteq A$). Since B already includes all the information contained within A, we decide to keep B (the maximal itemset, in this case) and remove A. The motivation behind using only maximal frequent itemsets is that the itemsets will likely contain all the required items that make up a rule, and it is better to have one larger frequent itemset rather than multiple smaller ones. In other words, the maximal frequent itemset already contains the same information that several smaller ones would.

Remove frequent itemsets without any target API usage. Developers tend to use multiple different libraries and frameworks in their code to accomplish different tasks. Our technique focuses on mining API usages of one or more target library or framework, and therefore, we are not interested in other API usages, unless they are somehow connected to the target API elements. Thus, we remove all frequent itemsets that do not have at least one usage of the target library API (i.e., use at least one element of the API).

Remove semantically incorrect frequent itemsets. Even though frequent itemsets contain relationships as string literals (e.g., *"Method hasParam String"*, there is usually an implicit semantic connection between them. In other words, when ordered correctly, items in frequent itemsets should accurately describe usage rules. For example, if a frequent itemset contains a relationship such as *"@Fallback hasParam String"*, we expect to see another relationship *"Method annotatedWith @Fallback"* that *declares* the `@Fallback` annotation before its parameter is used. The same applies to method and constructor parameters, as well as to the *"definedIn"* relationship that links annotation parameters to configuration files. A frequent itemset must contain relationships that form a semantically correct sequence of actions for a given usage. Let "__" be a placeholder for any entity, *"@A"* be some annotation, and *"P_a"* be some annotation parameter. In a semantically valid frequent itemset, if there is a *"P definedIn __"* relationship, then there must *"@A hasParam P"* relationship; if there is a *"@A hasParam __"* relationship (no matter how many annotation parameters are there), then there must be the *"__annotatedWith @A"* relationship. If a frequent itemset violates any of the two mentioned implications, then the frequent itemset cannot be further used to generate valid candidate rule and therefore, we remove such frequent itemset.

Remove required annotation parameters within frequent itemsets. When declaring annotations, a framework or library can specify whether an annotation parameter is mandatory or optional. The difference between mandatory and optional parameters is in the presence of default value. If there is a default value in the parameter declaration, then developers can use the annotation without providing their own value. However, if developers use a mandatory annotation, but forget to provide a concrete value, then an exception is thrown due to the missing value of the parameter. Since the compiler already checks whether there are values for required (mandatory) annotation parameters, relationships of the form *"@A hasParam P"*, where $@A$ is some annotation and P is a *required* parameter, are useless. In other words, such relationships will yield rules that the compiler already checks. On the other hand, if P is optional, we keep such relationship in the itemset because there might be an implicit rule that somehow involves P. Note that this heuristic helps shrink the size, and not the total number, of frequent itemsets, making the generated rules more precise.

Remove frequent itemsets of size < 2. Maximal frequent itemsets of size 1 do not make sense because they have only 1 relationship. For example, a frequent itemset {*"Method annotatedWith @Asynchronous"*} may be interpreted as "all methods must be annotated with @Asynchronous" which is obviously false. In addition, such frequent itemsets cannot be used for candidate rule generation because a candidate rule is of the form "if-then" and thus, contains at least 2 items (relationships). Frequent itemsets of size 1 are not going to produce anything meaningful, so we remove such frequent itemsets.

Note that we apply the abovementioned heuristics in the order described. In addition to the built-in heuristics, one can provide custom, domain (or API) specific post-processing heuristics depending on the nature of the target API (e.g., see Chapter 5.1.5).

4.5 Generating Candidate Rules

After post-processing the frequent itemsets, we proceed to generating candidate rules. We introduce candidate rules in Chapter 2.3.

The candidate rules in the form of "If X, then Y" expressions naturally fit in with the usage rules we found in the documentation. They are simple to understand and can be easily translated into other rule forms, such as "X must have Y". Given a list of "if-then" rules, we know that an API usage is a violation of one (or more) rules if X condition holds true while Y does not.

For example, the rule below states that if a method is annotated with `@Asynchronous`, then the method should return `java.util.concurrent.Future`:

If *Method annotatedWith @Asynchronous*,
then *Method hasReturnType Future*

Our goal is to generate as few candidate rules as possible that contain as much information as possible within themselves. Based on our observations, sparser (smaller in size, but larger in quantity) candidate rules require more effort to confirm their validity. We therefore generate *one* semantically-correct candidate rule with the highest confidence per frequent itemset to avoid as many redundant candidate rules as possible.

We follow a two-step process: we first generate all possible candidate rules whose confidence is above the minimum confidence threshold. Then, we map each frequent itemset into a corresponding candidate rule and greedily select a rule (1) that has the highest confidence and (2) is semantically correct. If no such rule exists, we generate it ourselves while making sure rules we generate ourselves (i.e., not using FP-Growth) are semantically correct.

Generating only one rule per frequent itemset lowers the amount of redundant rules we have to work with by about 75% (for every frequent itemset, there are 4 candidate rules on average). A candidate rule is *redundant* if there exists another candidate rule that is essentially the same rule. Therefore, generating one semantically-correct association rule helps us retain only one "representative" rule per frequent itemset, instead of multiple similar, but redundant ones.

Chapter 5

RQ1: How effective is pattern mining in discovering annotation rules?

In this chapter, we evaluate our technique described in Chapter 4. Our goal is to investigate whether pattern mining techniques can be used to discover annotation usage rules in enterprise microservices libraries and frameworks, such as MicroProfile. Since the set of relationships (discussed in Chapter 4.1.3) is based on MicroProfile usage rules, we additionally evaluate our pattern mining technique on a similar framework for developing microservice applications in Java, called Spring Boot, to show the generalizability of the approach. *Spring Boot* is a framework extension of the Spring framework that is used for developing microservice applications in Java. We first present our evaluation setup and then the corresponding results that provide an answer to RQ1.

5.1 Evaluation Setup

To evaluate our pattern mining technique, we need input data in the form of client projects that use MicroProfile and Spring Boot, respectively. After cloning and parsing the projects and forming the input itemsets, we perform several pre-processing steps to ensure diversity of the mined rules. Our purpose is to mine as many unique rules as possible in the set of generated candidate rules, thereby decreasing the redundancy[1] of the generated rules.

[1]We provide definition of redundancy for candidate rules in Section 5.1.6

5.1.1 Step 1: Retrieving client projects

To mine usage patterns for MicroProfile and Spring Boot, we look for real-world industry-like client projects that use these libraries/frameworks. The more such projects we can find, the higher probability that we mine more diverse API usages and more likely to see enough usages to form a pattern.

To find client projects that use MicroProfile, we use a custom Python script that clones MicroProfile client projects from the MicroProfile GitHub repository's dependency graph[2]. A *depedency graph* is a graph that describes dependencies (repositories that the project depends on) and depedents (other repositories that depend on this project) of some project (repository) on GitHub. We focus on *dependents*, which are client projects that use MicroProfile. However, since the majority of the dependents have no stars, we filter out projects by their sizes. Our script scrapes the list of dependents page by page and focuses only on repositories that have size of at least 500 KB in an attempt to retrieve only real-world industry-level projects. Projects that have sizes less than 500 KB are more likely to be toy projects, e.g., a student's homework or a collection of unrelated code snippets that do not represent any real-world MicroProfile usages. To further increase the likelihood of fetching only real-world client projects, we ignore projects whose title contains any of the following keywords (case-insensitive): "demo", "workshop", "guide", "example", "playground", "getting-started", "sample", "starter", "quickstart", "quick-start", "tutorial". The projects that contain one of the mentioned keywords are more likely toy projects that might not have API usages that are representative of real-world projects. We also clone projects from the IBM organization's enterprise GitHub, which contains some closed source projects. We use query "import org.eclipse.microprofile" in the search and manually go through >100 repositories that the search returned in total. We focus on repositories that have projects that use MicroProfile and ignore repositories that match any of the keywords mentioned earlier. We retrieve about 40 proprietary IBM projects. In total, we clone 530 projects that use MicroProfile. We obtain

[2]`https://github.com/eclipse/microprofile/network/dependents`

usages (itemsets) from only 493 repositories. We could not properly parse the remaining projects due to syntax errors in code, so we skip these projects and do not create any itemsets from them.

To find client projects that use Spring Boot, we use the GitHub Search API. The Search API is a convenient tool to search for repositories that match user-specified criteria, such as maximum or minimum number of stars, last commit date, and whether the repository is a fork or not. The search criteria are specified in a query that the Search API uses to automatically find relevant repositories. In our query, we look for projects that match the keyword “import org.springframework.boot” and are in written Java. To ensure high quality of the projects (i.e., real-world projects, not some student’s homework or assignment), we focus only on repositories that have at least 100 stars, are not forks of any other repositories, and where the last commit was made after July 31, 2017. We set the minimum of stars to 100 because, unlike the lack of publicly available client projects that use MicroProfile, Spring Boot has a large number of client projects due to its popularity and earlier release than MicroProfile (Spring Boot and its underlying Spring framework were first released in 2014 [61] and 2004 [77], respectively; MicroProfile was first released in 2018 [43]). Similar to the process of searching for MicroProfile projects, we ignore projects with keywords in their titles that might indicate a toy project. Using the mentioned criteria, we automatically find and clone a list of 252 projects.

In addition to the list of 252 Spring Boot client projects obtained automatically through the GitHub Search API, we manually search for real-world client projects that are built with a microservices style in Java using Google’s search engine. We use queries such as “large-scale open source spring boot projects” and “real-world spring boot microservices” to find references to publicly available Spring Boot client projects. We then add 37 real-world industry-level microservices projects mentioned in 3 relevant sources: the Stack Overflow post about big open-source Spring Boot projects [80], a blog post on “10+ Free Open Source Projects Using Spring Boot” [67], as well as the dataset of microservice-based systems [63]. Note that some of the projects we find here were already found using the GitHub search API. In total, we retrieve 289 projects that use Spring Boot, out of which 281 are parseable (no syntax errors that prevent parsing).

To summarize, we run our mining technique separately on 493 client projects that use MicroProfile and 281 client projects that use Spring Boot.

5.1.2 Step 2: Parsing client projects

Since all of our projects are written in Java, we use JavaParser [24] along with its symbol solver to parse the projects.

JavaParser is a parser for the Java language that provides each project's code as an Abstract Syntax Tree (AST). Using the ASTs, we create input itemsets with the tracked code relationships discussed in Chapter 4.1.3. JavaParser's symbol solver helps us resolve types and their declarations, thereby allowing us to retrieve fully-qualified names of types so that we differentiate types with the same name, but from different packages. For example, there is the `@Asynchronous` annotation from `javax.ejb` package of Enterprise JavaBeans (EJB), as well as the `@Asynchronous` annotation from `org.eclipse.microprofile.faulttolerance` package of MicroProfile. Coincidentally, these two annotations have the same name, but they originate from different sources and have different purposes.

5.1.3 Step 3: Pre-processing input itemsets

After processing ASTs with JavaParser into input itemsets containing the relationships we discussed in Section 4.1.3, we pre-process the itemsets before mining patterns from them. The following pre-processing steps allow us to balance the possible skewed data because some projects may overcontribute usages of only a few API annotations, whereas the rest of the projects may not have usages of such API elements at all.

Take one unique input itemset from each project. Frequent itemset mining is likely to yield common usages (idioms), while our goal is to ideally mine only usage patterns that encode specification rules. 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 balance the bias towards certain projects, we take one unique itemset per project. For example, if project A has a list of input itemsets `[ I1, I2, I3 ]`, where `I1` and `I2` are same (i.e., have the same items), we will keep `I1` and delete `I2`, ending up with `[ I1, I3 ]`. In other words, we are interested in patterns that span multiple projects rather than come mostly from one project.

After we take one input itemset from each project, we move onto partitioning the input itemsets by sub-API.

Partition input itemsets by sub-API. The enterprise microservices frameworks and libraries, such as Spring and MicroProfile (MP), spread their functionality into different components in the form of Java packages. For example, the main package for all MicroProfile APIs is `org.eclipse.microprofile`. However, if a developer wants to just use the MicroProfile Fault Tolerance API, they can use only the `org.eclipse.microprofile.faulttolerance` component (sub-package). These components, which we refer to as *sub-APIs*, are placed in sub-packages and provide a common functionality for a specific purpose, such as fault tolerance.

We observe that only a small fraction of sub-APIs are used by client developers most of the time and thus, the number of usages from each API is very different. For example, in the projects that we use to mine candidate rules from, there are more than 900 itemsets that use MP OpenAPI (i.e., `org.eclipse.microprofile.openapi`) sub-API. However, there are only 45 usages of the MP Reactive (i.e., `org.eclipse.microprofile.reactive`) sub-API in the same set of projects. The reasons behind this are that some sub-APIs were released later than others, and some are simply more applicable for a wide range of use cases than others (e.g., not all applications need to use reactive messaging functionality from MP Reactive). As a result, while some usages appear less commonly than others, they are likely to not qualify as patterns due to support threshold (i.e., their support is lower than the fixed threshold configured for all usages at once).

Due to imbalance in the number of usages per sub-API, we focus on mining each sub-APIs separately instead of the entire framework or library API. This means that we can set the support threshold differently for each sub-API. We partition a list of all input itemsets into smaller, disjoint lists. We leverage a hash map data structure that maps each sub-API name (as a string) to a list of input itemsets that have elements of that sub-API. There might be a case where one input itemset contains elements from two or more different sub-APIs. For example, itemset { *Method annotatedWith @Fallback, Class annotatedWith @Liveness* } represents a class that contains a method with annotation *@Fallback*, and the class itself is annotated with annotation *@Liveness*. *@Fallback* comes from the `org.eclipse.microprofile.faulttolerance` sub-API, but *@Liveness* comes from the `org.eclipse.microprofile.health` sub-API. In such cases, we replicate and copy the input itemset for both sub-APIs so that we do not lose information from mining one sub-API or the other. Note that we use *relative* support threshold for each sub-API instead of an absolute one. For example, the relative support threshold of 10% in a sub-API with 100 usages equals to the minimum support threshold of 10 usages, and in case of a sub-API with 1000 usages, that means the minimum support threshold for that sub-API is 100 usages. Unlike the absolute minimum threshold which is fixed for all sub-APIs, the relative threshold allows us to dynamically gauge the threshold based on the total number of input itemsets for each sub-API.

Mining sub-APIs individually, instead of all input itemsets together, allows for diversity of the mined patterns. Unlike mining all the API usages at once where some items from less popular sub-APIs might be overshadowed by more popular ones (due to relative support threshold), mining each sub-API separately provides an opportunity to mine patterns even from less popular sub-APIs. Since we use relative support threshold, the threshold used for less commonly used sub-APIs is less than the threshold used for more commonly sub-APIs, allowing us to extract patterns even from less common usages overall.

After taking one input itemset from each project and grouping the itemsets by sub-APIs, we now run frequent itemset mining for each sub-API separately.

5.1.4 Step 4: Mining

When mining the input itemsets that we created from each project, we set the *relative* minimum support threshold $s_{min} = 0.15$ which means that an itemset is frequent, if and only if it appears in at least 15% of transactions. For example, if there are 100 transactions in total (from all projects) and $s_{min} = 0.15$, then a frequent itemset has to appear in 15 or more transactions. Note that this is done per sub-API meaning that absolute threshold will change according to the number of usages of each sub-API. We choose 15% 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, and values higher than that lead to very few frequent itemsets.

5.1.5 Step 5: Post-processing frequent itemsets

While we already describe some of built-in heuristics for post-processing frequent itemsets applicable to any annotation-based framework/library (described in Chapter 4.4), we provide one additional heuristic here that is applicable to MicroProfile and Spring Boot[3].

After we get frequent itemsets, we automatically remove the ones that have irrelevant sub-APIs. We first pre-define the sub-APIs we want to focus on because some sub-APIs may simply be internal sub-packages. There is a chance (we intend to minimize this chance by filtering out irrelevant projects) that there exists a project (e.g., an internal MicroProfile project) that defines non-public packages for internal usage, e.g., for testing. For example, while `org.eclipse.microprofile.faulttolerance` package is a MicroProfile sub-API designated for public use, the package `org.eclipse.microprofile.system` is not because the latter is used internally and not supposed to be imported by client developers. After we define what sub-APIs (packages) we want to focus on, we analyze every item within each itemset to resolve the origin of some

[3]Note that the heuristics described here may or may not be applicable to other Java library or framework APIs.

entity (e.g., annotation). If there is at least one item that belongs to a sub-API of interest, we keep the frequent itemset. In other words, if a frequent itemset contains usages of annotations or other code elements that do not belong to any of the sub-APIs of interest, then we remove the itemset.

Since Spring Boot is a framework extension, applications that use Spring Boot also use the core Spring framework sub-APIs. For Spring Boot, we focus on 19 sub-APIs that provide core functionality for developing microservices applications in Java, such as interacting with relational databases with JDBC using the "org.springframework.jdbc" package and manipulating Java beans using the "org.springframework.beans" package. For MicroProfile, we focus on 10 sub-APIs that provide variety of functionality for building and maintaining microservices using MicroProfile, such as the "org.eclipse.microprofile.health" and "org.eclipse.microprofile.graphql" packages. We include the full list of sub-APIs of MicroProfile and Spring Boot in Appendix A on page 86.

In addition to the built-in heuristics (discussed in Chapter 4.4) that remove semantically incorrect frequent itemsets, the post-processing steps discussed earlier cut down the number of frequent itemsets by about 25% (given relative support threshold = 15%, we get 75 final, post-processed frequent itemsets instead of 100). In fact, removing semantically incorrect frequent itemsets alone cuts down the number of frequent itemsets by about 21%, thereby decreasing redundancy without losing information. Based on our manual analysis of the mined frequent itemsets, we do not lose any information because many frequent itemsets tend to be very similar to each other (i.e., have a set of items that differ only by one or two elements) and removing some of them did not reduce the number of semantically unique candidate rules.

5.1.6 Step 6: Confirming candidate rules with domain experts

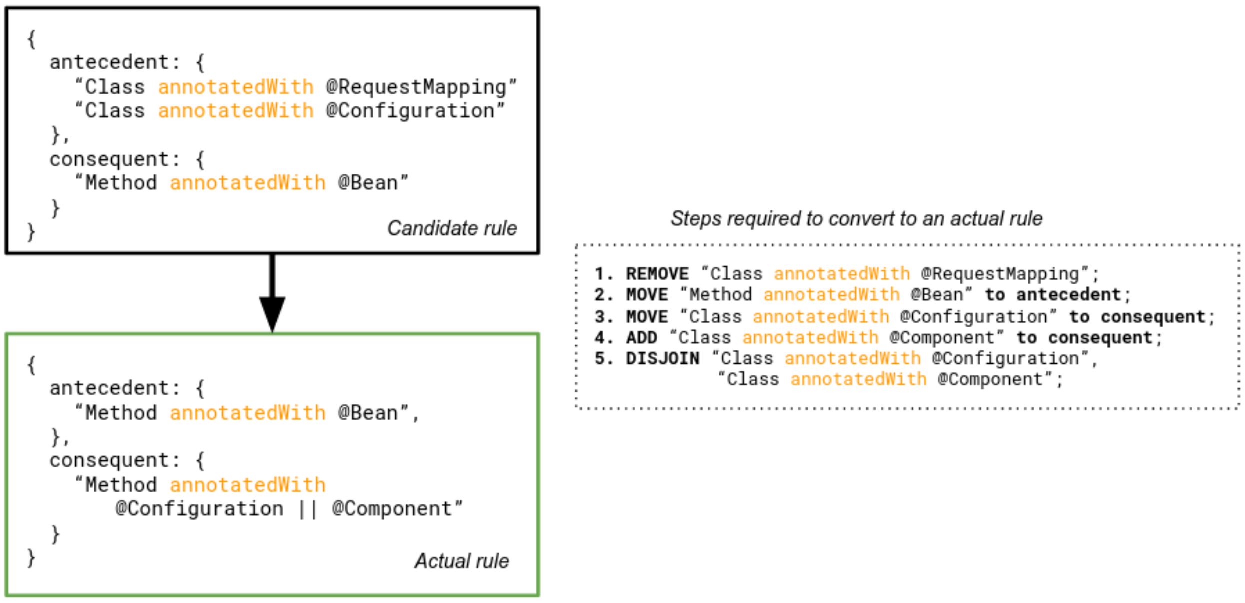
After applying post-processing steps and generating the candidate rules, we need to assess how close the candidate rules are to being the actual rules. Since we do not know the actual rules beforehand, we present our candidate rules to some of our collaborators who are domain experts in MicroProfile and Spring

Boot. For MicroProfile, we present the MicroProfile candidate rules to one of our collaborators from IBM, who is a direct contributor to the MicroProfile project. For Spring Boot, we present the Spring Boot candidate rules to two of our team members who have extensively used Spring Boot as client developers.

We ask the domain experts to confirm or deny rules with an option that they may suggest edits to convert a partially correct candidate rule into fully correct. A *partially correct candidate rule* is a rule that requires some editing to become an actual rule. The editing process involves the following operations: (1) the candidate rules contains redundant relationships that need to be removed; (2) lacks necessary relationships that need to be added; (3) is too restrictive so needs to be relaxed with "OR" relations that need to be added between two items ("AND" is the default relation between all items inside the antecedent or consequent); or (4) has relationships that need to be moved between the antecedent and consequent. To calculate how useful the candidate rules are to deriving actual rules, we introduce the following four operations for rule editing (where X is a relationship (item) in a candidate rule):

- **REMOVE** X**.** This operation removes a redundant item from a candidate rule thereby addressing point (1) above.
- **MOVE** X**.** This operation moves an item from the antecedent to consequent, or vice versa, thereby addressing point (4) above.
- **DISJOIN** X, Y(i.e., join with "OR"). This operation establishes an "OR" relation between two items, thereby addressing point (3) above.
- **ADD** X**.** This operation adds a new item to a candidate rule, thereby addressing point (2) above.

Using the 4 operations above, we calculate the edit distance between a candidate rule and an actual rule. Assuming that each edit operation costs 1 unit, the *edit distance* is the sum of all the edit operations suggested by our experts for some candidate rule. For example, in Figure 5.1, the edit distance between the candidate and actual rule is 5 because it requires 5 edit



```
{
  antecedent: {
    "Class annotatedWith @RequestMapping"
    "Class annotatedWith @Configuration"
  },
  consequent: {
    "Method annotatedWith @Bean"
  }
}
```

```
{
  antecedent: {
    "Method annotatedWith @Bean",
  },
  consequent: {
    "Method annotatedWith
       @Configuration || @Component"
  }
}
```

```
1. REMOVE "Class annotatedWith @RequestMapping";
2. MOVE "Method annotatedWith @Bean" to antecedent;
3. MOVE "Class annotatedWith @Configuration" to consequent;
4. ADD "Class annotatedWith @Component" to consequent;
5. DISJOIN "Class annotatedWith @Configuration",
           "Class annotatedWith @Component";
```

Figure 5.1: Example of steps required to edit a candidate rule into an actual rule.

operations (1 **REMOVE**, 2 **MOVE**-s, 1 **ADD**, 1 **DISJOIN**). We use edit distance as a metric to calculate the effectiveness of pattern mining in the context of mining annotation rules. Intuitively, if most candidate rules have large edit distance (meaning that they require nearly an overhaul) to actual rules, then our pattern mining approach has little use because the candidate rules are not very useful, and in such case, it is not much different from writing the API usage rules manually from scratch. 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 an actual one.

Despite the fact we only generate one candidate rule per frequent itemset (see Chapter 4.5), there might still be redundancy among candidate rules. A *redundant* rule does not add a new piece of information and is basically a variant of some other existing rule. In other words, there may be multiple variants of candidate rules that intrinsically (semantically) refer to the same actual rule. We establish the redundancy metric R that measures the ratio of unique rules to redundant rules (i.e., how many unique rules are there among all candidate rules regardless of edit distance):

$$R = 1 - (C_{unique}/C_{total}) \tag{5.1}$$

The redundancy metric R ranges from 0% (no redundancy, all rules are semantically unique) up to 100% (fully redundant, all rules are variants of each other). For example, say we generate 3 candidate rules "*If A or B, then C*", "*If A or D, then C*", and "*If A or E, then C*". Assume that based on expert feedback, all of the 3 mentioned rules correspond to 1 unique actual rule, that is "*If A, then C*". Therefore, the redundancy is $R = 1 - 1/3 = 67\%$.

In summary, we use the edit distance and redundancy metrics as proxies to measure the effectiveness of pattern mining.

5.2 Results

We now present the results of our experiment for MicroProfile and Spring Boot projects, respectively, based on the setup mentioned in the section above.

5.2.1 Descriptive Statistics

MicroProfile. For MicroProfile, in terms of the input to our mining technique, we extract 33,852 raw input itemsets across all sub-APIs. We apply the de-duplication heuristic (i.e., the one that takes one unique itemset from each project) and keep only 17,612 input itemsets. We also filter out itemsets without any MicroProfile usage, thereby leaving us with 3,049 pre-processed input itemsets. Finally, we run our mining technique per sub-API on a total of 3,049 input itemsets.

In terms of the output of our mining technique for MicroProfile, we initially retrieve 101 raw frequent itemsets. After applying the post-processing heuristics discussed in the experiment setup (including the built-in ones in Chapter 4.4), we get 77 final frequent itemsets. Based on the final frequent itemsets, we generate 77 candidate rules. It took our implementation about 5 minutes from reading the raw input (client projects) to generating the final output (candidate rules).

Spring Boot. For Spring Boot, in terms of the input data, we extract 52,690 raw input itemsets. After removing the duplicate itemsets (i.e., taking one unique itemset from each project), we have 26,402 input itemsets. We also remove itemsets that do not have any usages of Spring (Spring Boot is simply an extension of the Spring framework) and end up with 40,695 pre-processed input itemsets. Note that the total number of itemsets across all sub-APIs increases from 26,402 to 40,695 because we create copies of the same itemset for each sub-API for the reasons discussed in Step 3 in Chapter 5.1. We then run our mining technique on the 40,695 final input itemsets.

In terms of the output data for Spring Boot, we get 62 raw frequent itemsets. After applying the same post-processing steps we used for MicroProfile (though with different set of concrete sub-APIs), we get 54 final frequent itemsets. Based on these frequent itemsets, we generate 54 candidate rules. The overall process from reading the client projects to generating the rules took about 8 minutes.

Discussion of descriptive statistics. Overall, we extract more input itemsets from the Spring client projects than the MicroProfile ones, even though we have fewer client projects for Spring than MicroProfile. This might be due to the fact that the Spring Boot projects that we use are on average larger than the MicroProfile projects.

Tables 5.1 and 5.2 show the number of itemsets, as well as number of distinct projects each sub-API appears in (for Spring and MicroProfile, respectively). In the best case, usages of all sub-API should appear in all projects (irrespective of the volume of usages, i.e., how pervasive the usages are in one project), respectively, thereby increasing the likelihood of mining general usage specifications across all projects. In Spring, while some sub-APIs appear in numerous projects (such as the top 5 that appear in at least 100 different client projects), others do not have any usages at all (e.g., *org.springframework.shell*). Similarly, but not surprisingly, the top 5 sub-APIs by the number of projects also are the top 5 by the number of itemsets (8255, 6022, 5791, 4246, 1521, respectively). We see similar behavior with MicroPro-

Sub-API	**# of itemsets sub-API appears in**	**# of projects that have sub-API usage**
org.springframework.web	8255	257
org.springframework.beans	6022	257
org.springframework.context	5791	256
org.springframework.boot	4246	224
org.springframework.security	1521	101
org.springframework.http	677	90
org.springframework.core	570	96
org.springframework.jdbc	90	41
org.springframework.jms	36	7
org.springframework.aop	25	22
org.springframework.messaging	64	21
org.springframework.orm	20	8
org.springframework.retry	6	3
org.springframework.ldap	5	1
org.springframework.remoting	3	2
org.springframework.expression	1	1
org.springframework.shell	0	0
org.springframework.tx	0	0
org.springframework.asm	0	0

Table 5.1: Breakdown of sub-API usages in Spring.

Sub-API	**# of itemsets sub-API appears in**	**# of projects that have sub-API usage**
org.eclipse.microprofile.openapi	702	85
org.eclipse.microprofile.config	582	217
org.eclipse.microprofile.metrics	577	132
org.eclipse.microprofile.rest	324	68
org.eclipse.microprofile.faulttolerance	279	63
org.eclipse.microprofile.health	156	77
org.eclipse.microprofile.graphql	139	6
org.eclipse.microprofile.jwt	127	6
org.eclipse.microprofile.opentracing	70	22
org.eclipse.microprofile.reactive	38	18

Table 5.2: Breakdown of sub-API usages in MicroProfile.

file sub-APIs where the top 3 sub-APIs by the number of projects they appear in are also the top 3 sub-APIs by the number of itemsets they appear in.

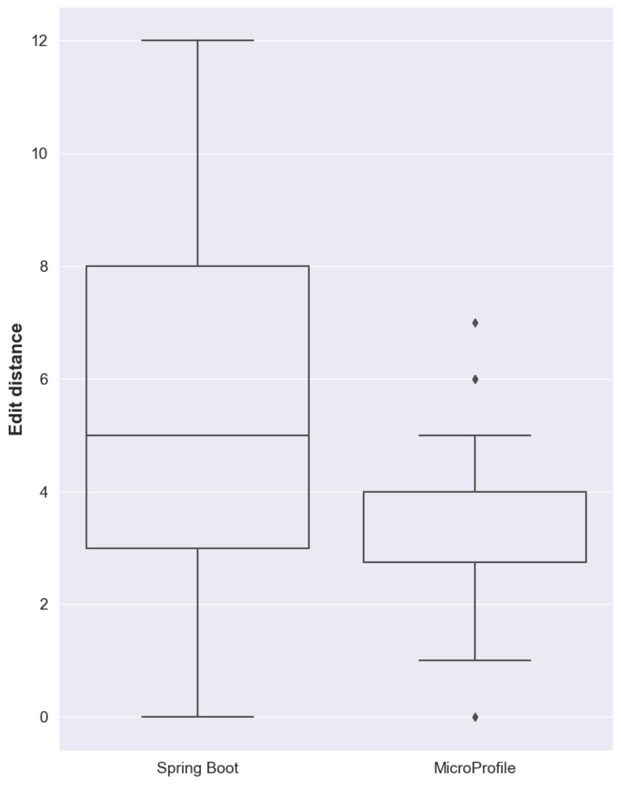


Figure 5.2: Distribution of edit distance for Microprofile and Spring candidate rules.

5.2.2 Effectiveness

Overall, we retrieve 77 candidate rules for MicroProfile and 54 candidate rules for Spring Boot, respectively. Based on our initial review of the rules, as well as expert feedback from our collaborators and team members, 36 out of 77 MicroProfile candidate rules need some editing to be done to turn into actual rules. The rest of the candidate rules are not potentially any rules. For Spring Boot, there are 37 out of 54 candidate rules that require editing. We provide a list of all mined rules for both frameworks in our publicly accessible GitHub repository [59].

Figure 5.2 shows edit distance distribution for Spring Boot (left box) and MicroProfile (right box) projects for the 37 and 36 rules, respectively. In MicroProfile, for the candidate rules that require some edits, one has to perform 3.61 edit operations on average. The median edit distance is 4.0. For Spring Boot's candidate rules, one has to perform 5.0 edit operations on average. The

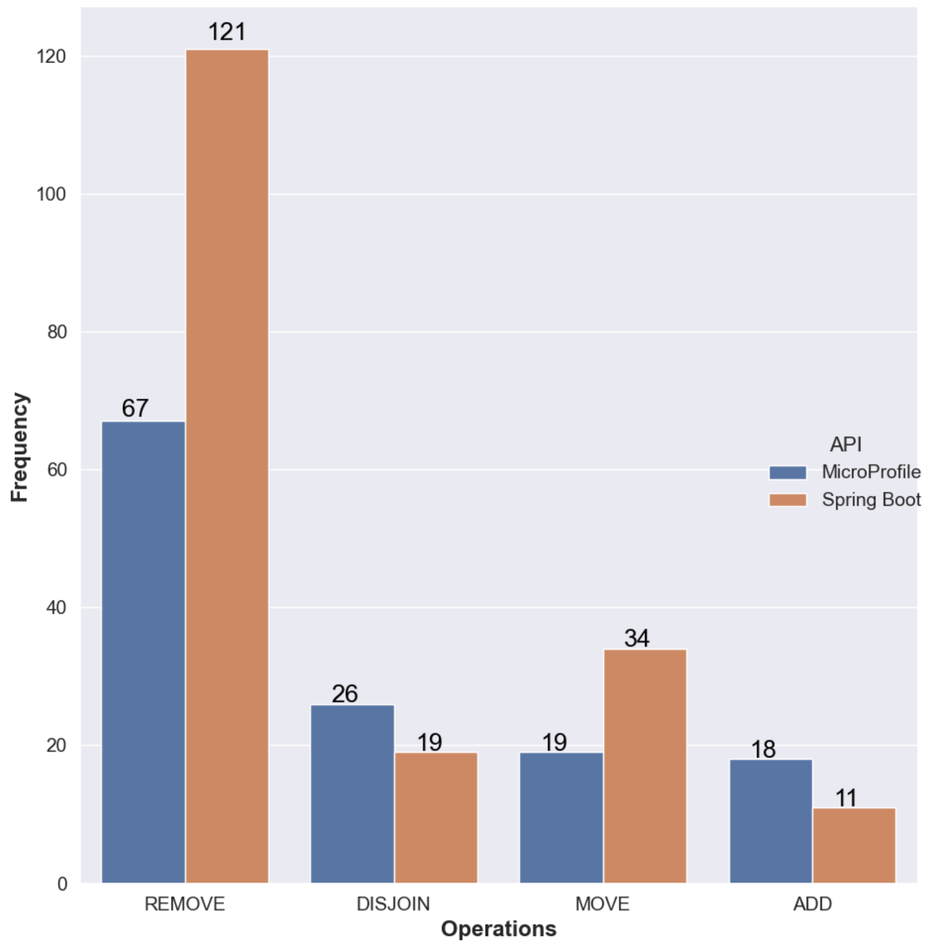
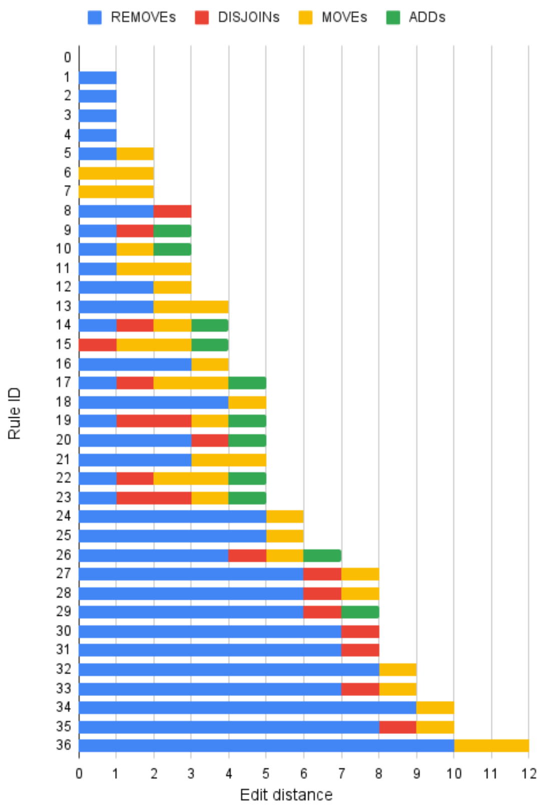


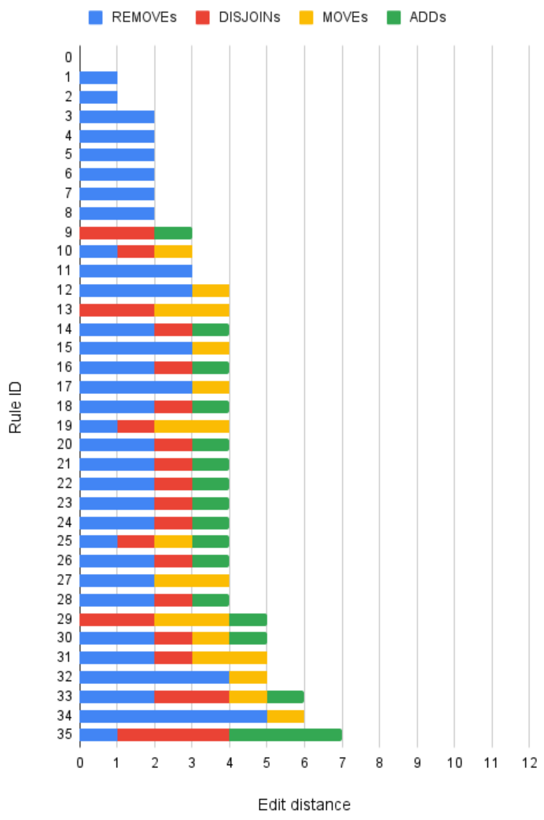
Figure 5.3: Total frequency of each operation across all rules in MicroProfile and Spring APIs, respectively.

median edit distance for Spring Boot candidate rules is 5.0. Both Spring Boot and MicroProfile contain only 1 candidate rule that is an actual rule and thus, does not require any editing (i.e., edit distance is 0). Furthermore, even though Spring Boot generates less candidate rules, the edit distance is higher than for MicroProfile candidate rules. Overall, 75% of the candidate rules require 8 or less edits, while there exists a rule that needs 12 editing operations.

We now report the types and distribution of operations needed to edit the candidate rules. When editing, not all operations are needed equally likely. Figure 5.3 shows grouped barplots of edit distance operations in MicroProfile (blue) and Spring Boot (orange) projects. It is clear that the **REMOVE** operation is the most popular edit operation due to pervasiveness of redundant

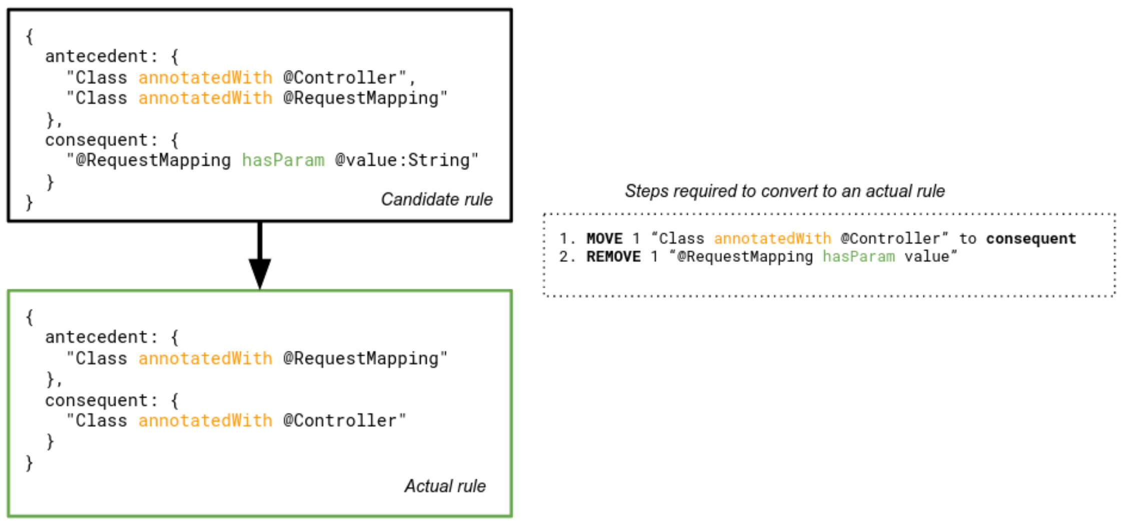


(a) Spring Boot candidate rules.



(b) MicroProfile candidate rules.

Figure 5.4: Breakdown of edit distance of all candidate rules per API



```
{
  antecedent: {
    "Class annotatedWith @Controller",
    "Class annotatedWith @RequestMapping"
  },
  consequent: {
    "@RequestMapping hasParam @value:String"
  }
}
```

```
1. MOVE 1 "Class annotatedWith @Controller" to consequent
2. REMOVE 1 "@RequestMapping hasParam value"
```

```
{
  antecedent: {
    "Class annotatedWith @RequestMapping"
  },
  consequent: {
    "Class annotatedWith @Controller"
  }
}
```

Figure 5.5: An example of a mined candidate rule (Spring Boot) with required edits based on expert feedback.

items within candidate rules. In addition, Figure 5.4 depicts edit distance distribution by edit operations in each mined and confirmed rule for both Spring Boot and MicroProfile APIs. It shows that the **REMOVE** operation comprises a large portion of the distribution of edit operations in Spring Boot and less so in MicroProfile. For example, Figure 5.5 depicts one of the mined Spring Boot candidate rules that needs 2 edit operations, one of which is the removal of the `value` parameter which is optional. The least needed operation is the **ADD** operation implying that our pattern mining approach captures most of items necessary to make up an actual rule. However, we cannot capture some items due to limitations in our approach. For example, while the rule with ID 8 from Table 5.3 is correct, it is not complete, i.e., it lacks a *"Param_-name definedIn <Environment properties>"* relationship, implying that the *name* parameter should be defined as an environment variable. To add such a relationship, we have to check the environment in which an application is running (i.e., at runtime) and that requires dynamic analysis. Since our mining approach is based on static analysis, we do not check for environment variables and thus, cannot track such a relationship.

In addition to editing the rules themselves, there is some redundancy in cases of both APIs. In the set of generated and confirmed candidate rules

ID	Antecedent	Consequent	Implications when violated
1	Field *annotatedWith* @ConfigProperty	Field *annotatedWith* @Inject	NullPointerException (which leads to Internal Server Error 500)
2	Class *annotatedWith* @RegisterRest-Client	(Method ∥ Class) *annotatedWith* @Path	Faulty behavior without explicit error (HTTP 404 Not Found)
3	Method *annotatedWith* @Incoming	(Class *annotatedWith* @Application-Scoped) ∥ (Class *definedIn* server.xml)	Faulty behavior without explicit error (empty list instead of list with objects)
4	Method *annotatedWith* @Outgoing	(Class *annotatedWith* @Application-Scoped) ∥ (Class *definedIn* server.xml)	Faulty behavior without explicit error (empty list instead of list with objects)
5	“Method *hasParam* Param_T”, “Param_T *annotatedWith* @Path-Param”	(Method ∥ Class) *annotatedWith* @Path	Faulty behavior without explicit error (simply gets ignored or causes 404)
6	Field *annotatedWith* @Claim	Field *annotatedWith* @Inject	NullPointerException (which leads to Internal Server Error 500)
7	Method *annotatedWith* @Query	Class *annotatedWith* @GraphQLApi	Faulty behavior without explicit error (GraphQL interface on ‘/graphql‘ is not accessible
8	“Field *annotatedWith* @ConfigProperty”,“@ConfigProperty *hasParam* Param_name:String”	Param_name:String *definedIn* Config-File_microprofile-config.properties	DeploymentException (which leads to Internal Server Error 500)
9	Method *annotatedWith* (@GET ∥ PUT ∥ POST ∥ UPDATE ∥ DELETE)	(Method ∥ Class) *annotatedWith* @Path	Faulty behavior without explicit error (HTTP 404 Not Found)
10	Field *hasType* JsonWebToken	Field *annotatedWith* @Inject	NullPointerException (which leads to Internal Server Error 500)
11	Field *annotatedWith* @RestClient	Field *annotatedWith* @Inject	Faulty behavior without explicit error (HTTP 404 Not Found)
12	Class *annotatedWith* @Health	Class *implements* HealthCheck	Faulty behavior without explicit error (missing liveness feature/metric)
13	Class *annotatedWith* @Liveness	Class *implements* HealthCheck	Faulty behavior without explicit error (missing liveness feature/metric)
14	Class *annotatedWith* @Readiness	Class *implements* HealthCheck	Faulty behavior without explicit error (missing liveness feature/metric)

Table 5.3: Unique mined and confirmed candidate MicroProfile rules.

ID	Antecedent	Consequent	Implications when violated
1	Class *annotatedWith* @EnableConfigurationProperties	Class *annotatedWith* @Configuration	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
2	Class *annotatedWith* (@EnableTransactionManagement || @EnableJpaRepositories)	Class *annotatedWith* @Configuration	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
3	Class *annotatedWith* @RequestMapping	Class *annotatedWith* @Controller	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
4	Class *annotatedWith* @EnableRetry	Class *annotatedWith* @Component	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
5	Method *annotatedWith* @Bean	Class *annotatedWith* @Configuration	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
6	Class *annotatedWith* @EnableConfigurationProperties	Class *annotatedWith* @Configuration	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
7	Class *annotatedWith* @EnableWebSecurity	Class *annotatedWith* @Configuration, Class *extends* WebSecurityConfigurerAdapter	Runtime exception (IllegalStateException)
8	Field *annotatedWith* @Id	Class *annotatedWith* @Entry, Field *mustNotBeAnnWith* @Attribute	Runtime exception (IllegalStateException)
9	Class *annotatedWith* @EnablePrometheusEndpoint	Class *annotatedWith* (@Component || @SpringBootApplication)	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
10	Method *annotatedWith* @Retryable	Class *annotatedWith* @Service	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
11	Field *annotatedWith* @Autowired	Class *annotatedWith* @Component	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
12	Method *annotatedWith* @ConditionalOnMissingBean	Method *annotatedWith* @Bean	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
13	Method *annotatedWith* @DependsOn	Method *annotatedWith* @Bean	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
14	Method *annotatedWith* @Recover	Class *annotatedWith* @Service	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
15	Class *annotatedWith* @EnableJms	Class *annotatedWith* @Configuration	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
16	Class *annotatedWith* @Order	Class *annotatedWith* @Component	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)
17	Field *annotatedWith* @Attribute	Class *annotatedWith* @Entry	Faulty behavior without explicit error (class is not registered as bean, annotation ignored)

Table 5.4: Unique mined and confirmed candidate Spring Boot rules.

for Spring Boot, the 37 candidate rules that we mine and confirm correspond to only 17 unique confirmed rules. Thus, the redundancy for Spring Boot is $R = 1 - (17/37) = 54\%$, meaning there is about 2 variants of candidate rules that represent the same 1 actual rule. Similarly for MicroProfile, we only have 14 unique rules out of 36 confirmed ones leading to redundancy of $R = 1 - (14/36) = 61\%$ implying that about 6 out of 10 rules are simply variants of each other and represent the same actual rule. Tables 5.3 and 5.4 list all unique confirmed rules for MicroProfile and Spring Boot, respectively.

5.2.3 Findings

Our findings show that our pattern mining approach is feasible in automatically extracting annotation usage rules for both MicroProfile and Spring Boot. While the mined and confirmed rules have to be modified to some extent, our approach provides a good starting point for framework developers for easy modification into specifications.

RQ1: Our pattern mining approach is feasible and generalizable to similar microservices Java frameworks, and finds 14 and 17 unique confirmed candidate rules for MicroProfile and Spring Boot, respectively. Our approach provides a good starting point for framework developers for easy modification of the mined and confirmed rules into usage specifications because the mined rules require only 3 to 5 edits on average.

5.2.4 Implications

To the best of our knowledge, there is no study that applied pattern mining techniques for extracting annotation-based API usages rules. Based on our results, we find that our pattern mining approach is able to extract annotation usage rules, but these patterns require editing to derive actual rules. Some operations require minimal mental effort such as removing optional relationships, while others, such as the `ADD` operation, require more effort. In other words, library developers (or experts of some API, in general) do not need to think too much to know that some existing items are optional and need to be removed (thus, minimal intellectual effort is needed), but if something is missing from the rule, library developers have to think harder (and perhaps search in external sources and communicate with other developers) to add missing items into a candidate rule (thus, much more intellectual effort is needed). Based on our observations, the more “intellectually demanding” operations are needed less and the less “intellectually demanding” ones are needed more.

Chapter 6

RQ2: How common are violations of the mined annotation rules?

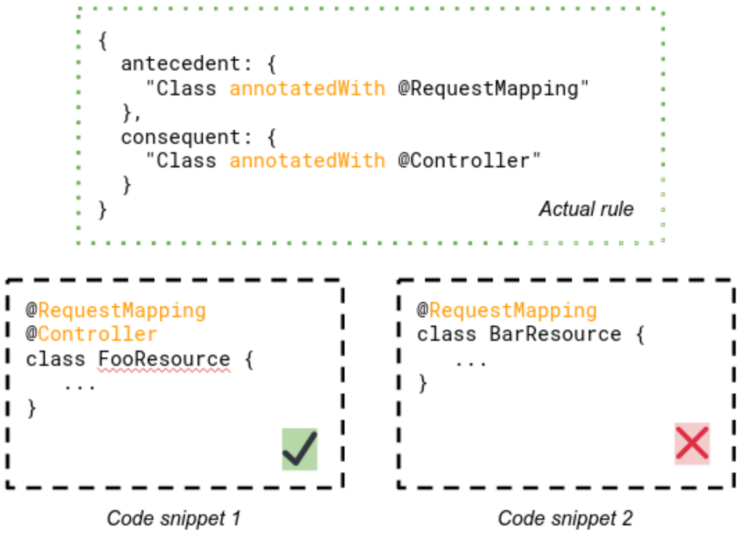
In this chapter, we aim to understand how widespread are violations of the confirmed candidate rules that we mined. The number of violations and their severity are the proxy for the usefulness of these rules in practice. We encode all the unique actual rules that we mined from MicroProfile and Spring Boot projects, respectively. We use JavaParser as our static analysis checker to scan code for violations, namely the locations in code where for any encoded rule, the antecedent is `true` but the consequent is `false`. For example, Figure 6.1 depicts a correct usage (code snippet 1) and a violation (code snippet 2) of one of the actual Spring Boot rules. When the static analysis checker for this rule encounters code snippet 2, it reports a violation stating that the class with annotation `@RequestMapping` is missing annotation `@Controller`.

6.1 Setup

We encode 14 unique candidate rules for MicroProfile and 17 rules for Spring as static analysis checkers, meaning that we scan code without running it.

6.1.1 Step 1: Fetching Projects

MicroProfile. For MicroProfile, we run our checkers on the same set of projects that we use for mining, because our set already contains all non-toy


```
{
  antecedent: {
    "Class annotatedWith @RequestMapping"
  },
  consequent: {
    "Class annotatedWith @Controller"
  }
}
```

```
@RequestMapping
@Controller
class FooResource {
   ...
}
```

```
@RequestMapping
class BarResource {
   ...
}
```

Figure 6.1: Encoded rule with correct and incorrect usage (violation).

projects we could find. We first scan the projects' latest commit for violations, and if we do not find any violations, we run our checkers on all relevent commits from the commit history. A commit is *relevant* if the commit change (text addition or removal) contains any of the elements (annotation names or types) of the encoded rules. These commits are likely to introduce or remove violations that we are interested in, and that way, we do not waste time analyzing irrelevant commits.

Spring Boot. For Spring Boot, we fetch a separate set of 1,139 projects that matched our search criteria. Having a different set of projects for scanning violations allows us to demonstrate generalizability. We fetch these projects using GitHub Search API. Our search criteria include projects that have at least 20 stars, are written in Java, and match the keyword "Spring Boot". We remove any projects that were in our previous list (i.e., the one used for mining).

6.1.2 Step 2: Encoding mined rules

MicroProfile. For MicroProfile, we encode 14 mined and confirmed candidate rules using JavaParser. With the help of JavaParser, we scan the code as ASTs and look for locations in code where, for all rules, the antecedent is `true` but the consequent is `false`.

Spring Boot. In a similar fashion, we encode 17 mined and confirmed candidate rules for Spring Boot and follow the same process as for MicroProfile rules.

6.1.3 Step 3: Usage issues in Stack Overflow (Spring Boot only)

Since Spring Boot is popular among the Java enterprise web application community, we look for manifestations of violations of the candidate rules in Stack Overflow. A violation manifestation is a Stack Overflow post where the question asks for help with any of the encoded rules, and the answer contains the consequent part of these rules. For example, say we want to find manifestations of violations of the following rule:

> *If a method is annotated with* `@Query`*, then the class containing the method should be annotated with* `@GraphQLApi`

To retrieve only relevant posts, we would first manually search for posts that contain both annotations `@Query` and `@GraphQLApi`. We thus use search queries such as “`"@Query" "@GraphQLApi"`” so that we match both annotations in a post. We then manually check whether each post contains the element (e.g., annotation) of the consequent (i.e., `@GraphQLApi`) in the accepted answer of the post. In other words, an SO post is a violation manifestation, if and only if the accepted answer states that the key element the author of the post is missing is the element which is part of the consequent. If there is no accepted answer, we choose the answer with highest number of votes among all the available ones. For example, given the SO post contains the `Query` annotation and the author explicitly asks about an information about this annotation, the answers such as *“Add @GraphQLApi ...”* or *“Missing @GraphQLApi ...”* are indicative of violation manifestations of the above rule.

Note that we do not search for usage rule violations of MicroProfile API in Stack Overflow because, as previously discussed in Chapter 4.1.1, there are no more than 300 questions and most of them are runtime-specific (opposed to MicroProfile-specific or general for all runtimes).

6.2 Results

6.2.1 Findings

MicroProfile. For MicroProfile, we scan for violations in the latest commit of all projects. However, since we do not find any violations, we scan for them in the commit history, i.e., we run our tool on every commit. We find 12 MicroProfile API usage violations of 6 distinct usage rules in 9 client projects that we manually confirm that they were fixed later in the commit history.

We report the consequences of violations of the mined rules in Table 5.3. Since client projects sometimes require extra work to build and run (e.g., finding and installing the necessary dependencies), we manually create a toy project and simulate a violation in that project. This allows to understand the consequences of violating rules and not rely on the ability of client projects to build and run. By simulating a violation of every rule, we observe two types of consequences: silent errors (which result in faulty behavior without the application crashing) and explicit runtime errors. Rules with ID 1, 6, 8, and 10 result in runtime exceptions when violated, while violations of the rest of the rules result in silent errors and faulty behavior. For example, when violation, the rule with ID 9 leads to the resource class (the class that is missing the `@Path` annotation) not being picked up by the runtime and registered as an HTTP resource. This then results in HTTP 404 “Not Found” because the resource class is not registered with the runtime due to the missing annotation.

Among the violations in client projects, there two types of consequences of violating the 6 rules: violations of 3 rules manifest as runtime exceptions and violations of the other 3 manifest in silent, faulty behaviors which do not throw result in any explicit error. However, the rules with violations which result in explicit error (e.g., runtime exceptions) do not have descriptive error messages. For example, while the rule with ID 6 (see Table 5.3) causes a `NullPointerException` (runtime error) when violated, it does not explicitly say that the `@Inject` annotation is missing on the field. Further, developers may struggle debugging the 4 rules that do not have explicit errors when violated. For example, the rule with ID 7 results in faulty behavior that does not crash an application when violated (i.e., there is no `@GraphQLApi` on the class). There is no explicit exception when the rule is violated. We observe is that it returns HTTP code 404 "Not Found" when trying to access the GraphQL query interface on HTTP path `/graphqlapi`. While such behavior seems frivolous, it may have more serious consequences, such as developer not being able to figure out why they are not able to render the web interface or causing other services that rely on GraphQL queries to crash.

We also find another set of 8 potential violations, but could not confirm whether they are true violations because the code that contained them was entirely deleted (e.g., a file containing the violation was deleted).

Spring Boot. For Spring Boot, we scan for violations in the latest commit and find 28 violations of 4 distinct usage rules in 24 projects.

To find out the consequences of violating the mined and confirmed rules, we follow the same process as described above. We list all the rules and corresponding violation consequences in Table 5.4. In fact, only 2 rules result in `IllegalStateException` (runtime error) when violation. The rest of the rules result in silent, faulty behaviors, similar to consequences of violating the confirmed MicroProfile rules.

Among the violations of 4 confirmed rules in client projects, the rule with ID 11 is the only one that causes a runtime exception when violated. Violations of the other 3 rules (rules with ID 3, 5, and 16) do not cause any explicit errors and instead have a similar silent, faulty behavior such as the rule with `@GraphQLApi` (see above). For example, violating the rule with ID 5 means that the class that is missing `@Component` or `@Configuration`. When the class is not annotated with any of the two annotations, then it is going to be ignored by the Spring Framework runtime and is essentially a dead code if not used anywhere else in the codebase. In such cases, it is not clear whether it is a dead code by mistake or on purpose (e.g., developers wanted to ignore the class).

For additional confirmation, we report 28 violations as issues on the corresponding GitHub repositories. As of August 25, 2021, only 1 repository replied with the translated message (from Mandarin) equivalent to "Yes, it is a violation, but the class is not used.", meaning that one of the reported violations is true, but is intentional. While we have not received any response from the rest of the projects where the violations were found, the owners may have similarly intentionally not annotated the classes with one of the required annotations because they want purposefully to ignore the class for whatever reason.

Spring Boot in Stack Overflow. Using the queries that we mentioned in the setup earlier (Step 3), we find 9 Stack Overflow posts that are the manifestations of violations of the 5 mined rules. In fact, out of these 5 rules, we find violations of 2 in client projects (rules with ID 3 and 11), while the rest of the rules (rules with ID 4, 7, and 8) are not violated there. For example, one of the posts titled "@RequestMapping stopped working after adding DB and security features" experienced an issue with non-responsive `@RequestMapping` annotation, i.e., it did not work as intended. The post refers to the rule with ID 3 in Table 5.4. The attached execution logs are not useful because they do not contain any error, which conforms to our description of violating this rule. According to the accepted answer (by the owner themselves), the owner was simply missing the `@Component` annotation on the *Validator* class, the snippet of which was not even shared in the post indicating that debugging such issues may not be straightforward.

RQ2: We find that client developers violate 6 out of 14 encoded MicroProfile rules and 4 out of 17 encoded Spring Boot rules. Based on our observations with usages of Spring Boot in Stack Overflow, we additionally find manifestations of violations of 5 rules indicating that developers sometimes struggle to adhere to the implicit usage rules.

6.2.2 Implications

Our findings show that the violations of some of the mined and confirmed rules are common in the client projects of MicroProfile and Spring Boot. The findings also show that the client developers sometimes struggle to correctly use some Spring Boot annotations, thereby unintentionally violating usage rules. Even though the usage rules that we mine and confirm are simplistic (i.e., do not involve data and control flow), client developers do sometimes violate them, and the violations may lead to bugs and faulty behavior that may be difficult to debug without additional tooling. Therefore, the findings provide a motivation for tools that can assist client developers in correctly using the APIs thereby preventing annotation-based API misuses.

Chapter 7

Threats to Validity

7.1 Internal Validity

There may be candidate rules that we confirmed to be correct, but are actually not rules. However, such case is very unlikely given that we confirm candidate rules simultaneously with documentation and our collaborators. One of our industry collaborators is a direct contributor to MicroProfile who has actually written some of the documentation. For Spring Boot, we manually verified our rules in the documentation, as well as asked two of our team members to confirm the rules. For every confirmed rule, we find direct evidence in the documentation in the form of a web link. The combination of verifying the validity rules in the documentation and with our collaborators helps us minimize confirmation bias, so it is unlikely that a candidate rule that is confirmed to be correct is not actually true.

There may be bugs or incorrect behavior in our implementation of the mining approach. We create a set of synthetic examples for all relationships (discussed in Chapter 4.1.3) and test our tool to make sure it is able to correctly mine what we intend to mine.

We have created Python scripts to automatically calculate total edit distance, as well as visualizations related to edit distance operations. To make sure our tooling is correct, we get several samples of candidate rules and their edit comments, and compare the results.

7.2 External Validity

We focus only on Java APIs that provide functionality for building microservices and that are primarily used through annotations. We limit what kind of client projects we focus on (ignoring toy projects) while having access mostly only to open source projects. However, in addition to open source projects, we have access to and add more than 40 closed-source IBM projects to the set of projects we use. We show that our approach is generalizable for libraries or frameworks for building enterprise microservices applications in Java by evaluating our approach on Spring Boot projects. We fetch more than 400 projects for each library/framework API, confirm our usage rules with domain experts, as well as find violations of the mined rules in client projects as well as Stack Overflow. However, the web applications that use MicroProfile and Spring Boot, especially the industry-level ones, tend to be closed-source. While we add closed-source IBM projects that use MicroProfile, our approach may also not generalize for frameworks or libraries that provide functionality other than building enterprise web applications, or the ones that are not primarily used through annotations or other form of metadata.

7.3 Construct Validity

When measuring the effectiveness of pattern mining for extracting annotation usage rules, we use number of semantically unique confirmed candidate rules, redundancy, and edit distance. The number of confirmed candidate rules serves as an indicator whether pattern mining actually mines useful information or not. We encode rules into static checkers to find violations in client code. But we do not find violations of some rules. One of the possible explanations is that the violations of these rules are probably done early in the development process. For example, developers make a mistake that results in usage violation, but then detect it by running (deploying) the application locally thereby fixing it before the error seeps into production (or even gets recorded in a commit). In fact, we detect violations of some rules only through commit history which show that developers sometimes make a mistake in some commit and fix it in another commit later.

The redundancy and edit distance metrics measure how much information (among and within rules, respectively) is useful and how much is redundant. Edit distance is a proxy for usefulness of candidate rules to become real rules. However, we did not run a user study to determine how client developers would use it, and whether they would consider the rules (as static checkers) themselves useful or not.

Chapter 8

Discussion

We discuss the implications of our results, limitations of our approach, as well as next steps in mining annotation rules in enterprise Java applications.

8.1 Implications

In Chapter 5, we mine and confirm 14 and 17 unique candidate rules. We also find that it takes 3 to 5 edits, on average, to convert a candidate rule into a fully correct one. In Chapter 6, we scan for violations in real client projects and find violations of 6 and 4 MicroProfile and Spring Boot rules, respectively. Based on the results discussed in these two chapters, we find that pattern mining is feasible for extracting annotation-based API usage rules. In addition, we find that the mined and confirmed usage rules have corresponding violations in both real-world client projects of the APIs they use and Stack OVerflow (as violation manifestations). These results demonstrate that while violations of some annotation-based rules are common, it is not quite clear whether the violations are intentional or not. The violations nevertheless have consequences that result in runtime exceptions or silent faulty behaviors, thereby showing a need for tools that could assist client developers in correctly using annotation-based APIs.

The results also show that pattern mining provides a starting point for framework developers for easy modification of the mined rules into usage specifications. Even though most of the mined rules need some editing before turning into complete usage specifications, they require only a few (3 to 5) edits to become fully correct rules. The need for editing for majority of rules by API experts implies that there is a need for a tool or interface that would allow API experts to easily confirm, deny, or modify the rules.

While we mine and confirm simplistic, yet useful rules, we do not mine more complex usage rules due to limitations in our approach, which we discuss next.

8.2 Limitations

We now discuss the limitations of our approach. Our approach does not mine some rules that involve relationships across code elements, such as methods and fields. Consider the following rule:

> *If there is a class with a method (e.g., `foo()`) that is annotated with `@Fallback` and `@Fallback` has parameter `fallbackMethod` with value `"bar"`, then the class should have another method named `bar()`, parameters and return type of which should be the same as in method `foo()`.*

In Listing 8.1, we have two methods, one of which is a *fallback* method for `getEntry` method. The rule above states that the value of annotation parameter `fallbackMethod` 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). Accordingly, both methods `getEntry` and `fallbackforGetEntry` have the same types of parameters and return type. Similar annotations and their usage rules exist in Spring, namely the annotations `@Retryable` and `@Recover`, both of which are equivalent to `@Fallback`.

We are not able to mine the rule above due to the following limitations. One of the limitations is that we are not checking annotation parameter values (e.g., a string literal value of `fallbackMethod` above) for their semantic meaning. In other words, a string literal value such as `"bar"` could mean multiple things, such as element or attribute in XML configuration files or a type, method, class, or even package in Java (code) files. In addition, we build

```java
import org.eclipse.microprofile.faulttolerance.Fallback;
import java.lang.String;

public class Foo {
    // If fails, method `fallbackforGetEntry()` will be called
    @Fallback(fallbackMethod="fallbackforGetEntry")
    public String getEntry(String id) {
        ...
    }

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

Listing 8.1: Specifying a fallback method

separate itemsets for fields, methods, and constructors, respectively. The primary reason behind using one itemset per construct (i.e., field, method, or class) is to avoid ambiguity when referring to method parameters. In other words, if we created one itemset per class for all methods, fields, and constructors together, we will not be able to differentiate which methods have which parameters, which is basically a limitation with frequent itemset mining. The rule above involves relationships that span two different methods. It is therefore expected that the rule mentioned earlier will not be mined because one itemset contains information about one method only and not several ones within one class.

8.3 Recommendations

To address the abovementioned limitation of our representation, future work can parse the parameter values and check whether the value is some method that exists within the same class or its parent. However, the more specific the tracked information is, the less likely the pattern will be mined. It is equivalent to tracking class names (too specific) instead of general class construct instead (general). Researchers can make an attempt to check whether a parameter value is not just a method, but also whether it is a field or a locally defined class.

Another possible direction is adapting our approach (i.e., the supported relationships as edges and the program entities as nodes) into simplified graphs without data and control flow (e.g., discussed in Chapter 3). The graphs are more likely to capture complex rules like the one in Listing 8.1. For example, one can initiate a graph from class entity with directional relationships, such as *"hasMethod"*, into methods, fields, and constructors, respectively, where these entities themselves have their own relationships, such as *"annotatedWith"*. Graphs store and track information separately per code element which provides a functionality to differentiate information for each code element, i.e., which method has what return type and parameter list. Unlike frequent itemsets that contain only unique relationships, graphs allow for duplicate relationships. However, based on our initial experiments, mining frequent graphs is a memory-intensive process which frequently leads to memory timeouts because mining graphs is based on *apriori mining* which looks at all possible (frequent) extensions of the graphs, generates the graphs, and repeats the process.

Another interesting direction is decreasing redundancy among candidate rules. As shown in Chapter 5, there is at least 50% redundancy among the generated candidate rules implying that there are two variants per one rule. The future work can look into a variety of techniques, ranging from clustering to active learning, that might help group very similar rules together in a group from which only one rule is generated. Decreasing redundancy is important, particularly when framework developers verify the generated rules. The more redundancy there is, the more time-consuming and error-prone the process of verifying rules becomes.

Chapter 9

Conclusion

In this thesis, we explore the idea of pattern-based discovery of Java annotation usage rules. Annotation usages are ubiquitous when it comes to using enterprise microservices frameworks. For example, MicroProfile and Spring Boot are two of multitude frameworks that provide annotations as their API. In collaboration with MicroProfile experts from IBM, we set out on a journey to verify annotation usage in MicroProfile applications.

We have examined related literature to find potential solutions for our needs. A general approach to verifying API usages in general is to write usage specifications, encode them into some tool (e.g., static analysis checkers) that will automatically validate a usage. Ideally, such usage rules should be automatically extracted that will at least provide a way for easier modification into usage specifications. We determine that there is no pattern mining tool or technique that supports mining of annotation usages.

To better understand what is necessary for mining of annotation usage rules, we first searched for possible usage rules of MicroProfile API. Based on information from the sources such as the official documentation, Stack Overflow, and the official MicroProfile forum on Google Groups, we determine 8 relationships that describe the code facts that we track in code. With these relationships in mind, we implement our approach to find usage patterns, that can be translated into usage specifications, using frequent itemset mining. Frequent itemset mining extract frequent itemsets which describe patterns of relationships that frequently occur in projects that use MicroProfile. Given frequent itemsets, we then generate patterns in the form of “if-then” candidate rules.

To evaluate our approach, we fetch and use 493 MicroProfile and 281 Spring Boot projects for mining. For MicroProfile, we extract 14 semantically unique usage rules which we confirm with our industry collaborators, one of whom is a direct contributor to MicroProfile. For Spring Boot, we extract 17 semantically unique usage rules which we confirm with documentation and our team members who have experience in using Spring Boot.

We also explore whether there are any violations of the mined and confirmed candidate rules. To find violations of the mined MicroProfile usage rules, we encode the rules into checkers and run them on the same set of projects that we used for mining. We find and manually validate 12 violations which were fixed later in the commit history. To find violations of Spring Boot API, we follow the same process, but use a different set of >1K projects that use the API. We find and manually validate 28 violations and report them as issues on the respective GitHub repositories. In addition to scanning Spring Boot projects for violations, we find manifestations of violations of the mined rules in 9 Stack Overflow posts.

Future work can investigate decreasing redundancy among candidate rules. A possible direction is investigating techniques that can decrease redundancy without losing useful information, such as clustering of similar rules together. Another issue is that most mined rules still need some editing to become correct. In future, we suggest researchers streamlining the process of confirming the rules with experts. A possible direction is to let the domain experts edit annotation-based rules via some user interface that can assist experts with confirming, denying, or modifying the rules. Ideally, static analysis checkers can then be automatically generated from confirmed rules. Then, such checks can be used by client developers to ensure they have no misuses in their code.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining api patterns as partial orders from source code: From usage scenarios to specifications,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 25–34.

[2] R. Agrawal, R. Srikant, *et al.*, “Fast algorithms for mining association rules,” in *Proc. 20th int. conf. very large data bases, VLDB*, Citeseer, vol. 1215, 1994, pp. 487–499.

[3] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, “A Systematic Evaluation of Static API-Misuse Detectors,” *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019.

[4] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, “Investigating next steps in static api-misuse detection,” in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR ’19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 265–275. DOI: `10.1109/MSR.2019.00053`. [Online]. Available: `https://doi.org/10.1109/MSR.2019.00053`.

[5] G. Ammons, R. Bodik, and J. R. Larus, “Mining specifications,” *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[6] Amos Kingatua, *13 Best Practices to Secure Microservices*, `https://geekflare.com/securing-microservices/`, [Online; accessed 14-July-2021], 2020.

[7] Apache TomEE, *Apache TomEE*, `https://tomee.apache.org/`, [Online; accessed 16-July-2021], 2021.

[8] 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.

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

[10] Danny Coward, *JSR 175: A Metadata Facility for the JavaTM Programming Language*, `https://www.jcp.org/en/jsr/detail?id=175#2`, [Online; accessed 16-July-2021], 2004.

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

[12] Eclipse Microprofile Contributors, *Annotation Type ConfigProperty*, `https://download.eclipse.org/microprofile/microprofile-2.0-javadocs-test/apidocs/org/eclipse/microprofile/config/inject/ConfigProperty.html`, [Online; accessed 16-July-2021].

[13] ——, *Different kinds of Health Checks*, `https://download.eclipse.org/microprofile/microprofile-health-2.1/microprofile-health-spec.html#_different_kinds_of_health_checks`, [Online; accessed 16-July-2021].

[14] ——, *Annotation Type LoginConfig*, `https://download.eclipse.org/microprofile/microprofile-2.0-javadocs-test/apidocs/org/eclipse/microprofile/auth/LoginConfig.html`, [Online; accessed 16-July-2021], 2021.

[15] Eclipse MicroProfile Fault Tolerance Contributors, *Eclipse MicroProfile Fault Tolerance, Asynchronous Usage*, `https://github.com/eclipse/microprofile-fault-tolerance/blob/master/spec/src/main/asciidoc/asynchronous.asciidoc`, [Online; accessed 16-July-2021], 2021.

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

[17] Emily Jiang, *Asynchronous (microProfile-fault-tolerance-api)*, `https://download.eclipse.org/microprofile/microprofile-fault-tolerance-3.0/apidocs/`, [Online; accessed 14-July-2021], 2020.

[18] M. Fähndrich, "Static verification for code contracts," in *Static Analysis*, R. Cousot and M. Martel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 2–5, ISBN: 978-3-642-15769-1.

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

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

[21] IBM Corp., *Open Liberty: An IBM Open Source Project*, `https://openliberty.io/`, [Online; accessed 16-July-2021], 2021.

[22] James Lewis, *Microservices*, `https://martinfowler.com/articles/microservices.html`, [Online; accessed 14-July-2021], 2014.

[23] P. Jashma Suresh, U. Dinesh Acharya, and N. Subba Reddy, "Study of effective mining algorithms for frequent itemsets," in *Intelligent Data Communication Technologies and Internet of Things*, Springer, 2021, pp. 499–511.

[24] JavaParser.org, *JavaParser - Home*, `http://javaparser.org/`, [Online; accessed 16-July-2021], 2021.

[25] Jonathan Barbero, *WebSphere Application Server support of MicroProfile*, `https://stackoverflow.com/questions/57530872/websphere-application-server-support-of-microprofile`, [Online; accessed 16-July-2021], 2021.

[26] Joshua Bloch, *A Brief, Opinionated History of the API*, `https://www.infoq.com/presentations/history-api/`, [Online; accessed 16-July-2021], 2018.

[27] JSR 339 Contributors, *JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services*, `https://jcp.org/en/jsr/detail?id=339`, [Online; accessed 16-July-2021], 2014.

[28] H. Kagdi, M. L. Collard, and J. I. Maletic, "An approach to mining call-usage patternswith syntactic context," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 457–460.

[29] 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*, 2010, pp. 117–126. DOI: `10.1109/CSMR.2010.20`.

[30] Kevin Sutter, *What's next for MicroProfile and Jakarta EE?* `https://www.eclipse.org/community/eclipse_newsletter/2018/september/mp_jakartaee.php`, [Online; accessed 16-July-2021], 2018.

[31] 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*, pp. 1–1, 2019.

[32] R. Lämmel and S. Peyton Jones, "Scrap your boilerplate: A practical design pattern for generic programming," vol. 38, Mar. 2003, pp. 26–37. DOI: `10.1145/604174.604179`.

[33] Leonardo Leite, *Wildfly 17 error "WFLYMETRICS0003: Unable to read attribute second-level-cache-hit-count" when statistics-enabled="true"*, `https://stackoverflow.com/questions/59322997/wildfly-17-error-wflymetrics0003-unable-to-read-attribute-second-level-cache-h`, [Online; accessed 16-July-2021], 2021.

[34] 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.

[35] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, Lisbon, Portugal: Association for Computing Machinery, 2005, pp. 306–315, ISBN: 1595930140. DOI: `10.1145/1081706.1081755`. [Online]. Available: `https://doi.org/10.1145/1081706.1081755`.

[36] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "Arbitrar: User-guided api misuse detection," 2021.

[37] Z. Li, A. Machiry, B. Chen, K. Wang, M. Naik, and L. Song, "ARBITRAR: User-Guided API Misuse Detection," in *IEEE S&P 2021*, 2021.

[38] D. Lo and S.-C. Khoo, "Smartic: Towards building an accurate, robust and scalable specification miner," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 265–275.

[39] D. Lo, G. Ramalingam, V.-P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," *Science of Computer Programming*, vol. 77, no. 6, pp. 743–759, 2012.

[40] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 501–510.

[41] 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, ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2015.01.004`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S016764231500012X`.

[42] S. Mehrpour, T. D. LaToza, and H. Sarvari, "RulePad: Interactive Authoring of Checkable Design Rules," *arXiv e-prints*, arXiv:2007.05046, arXiv:2007.05046, Jul. 2020. arXiv: `2007.05046 [cs.SE]`.

[43] MicroProfile, *MicroProfile: Optimizing Enterprise Java for a Microservices Architecture*, `https://microprofile.io/`, [Online; accessed 14-July-2021], 2021.

[44] MicroProfile Community and Google Groups, *[microprofile] MicroProfile 4.0 has been released*, `https://groups.google.com/g/microprofile/c/MgyHLEW58Nk`, [Online; accessed 16-July-2021], 2021.

[45] ——, *Loadbalancing for MicroProfile RestClient?* `https://groups.google.com/g/microprofile/c/sok2ojgo0gE`, [Online; accessed 16-July-2021], 2021.

[46] ——, *MicroProfile - Google Groups*, `https://groups.google.com/g/microprofile`, [Online; accessed 16-July-2021], 2021.

[47] ——, *MicroProfile 5.0 news and instructions*, `https://groups.google.com/g/microprofile/c/coOY1rxlSXE`, [Online; accessed 16-July-2021], 2021.

[48] ——, *MicroProfile Platform Specification discussion thread*, `https://groups.google.com/g/microprofile/c/RfcZFwdeQao`, [Online; accessed 16-July-2021], 2021.

[49] ——, *MicroProfile Steering Committee Scheduled for January 12th*, `https://groups.google.com/g/microprofile/c/3EUq808CwFk/m/-uYOnirVAAAJ`, [Online; accessed 16-July-2021], 2021.

[50] ——, *Startup probe qualifier name in MicroProfile Health*, `https://groups.google.com/g/microprofile/c/pYRQ4ZBuKfg`, [Online; accessed 16-July-2021], 2021.

[51] MicroProfile Community, Eclipse Foundation, *MicroProfile 4.0 Overview*, `https://docs.google.com/presentation/d/1p_gi3xLmPZsO11BKkC4QJwqjuaSGqhGZiqMcy`
[Online; accessed 14-July-2021], 2021.

[52] Microservices.io, Chris Richardson, *Who is using microservices?* `https://microservices.io/articles/whoisusingmicroservices.html`, [Online; accessed 14-July-2021], 2020.

[53] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *ECOOP 2010 – Object-Oriented Programming*, T. D'Hondt, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 2–25, ISBN: 978-3-642-14107-2.

[54] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, Mar. 2013, ISSN: 1049-331X. DOI: `10.1145/2430536.2430541`. [Online]. Available: `https://doi.org/10.1145/2430536.2430541`.

[55] 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)*, 2019, pp. 615–627.

[56] 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*, ser. ESEC/FSE '09, Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 383–392, ISBN: 9781605580012. DOI: `10.1145/1595696.1595767`. [Online]. Available: `https://doi.org/10.1145/1595696.1595767`.

[57] S. Nielebock, R. Heumüller, K. M. Schott, and F. Ortmeier, “Guided pattern mining for api misuse detection by change-based code analysis,” *ArXiv*, vol. abs/2008.00277, 2020.

[58] O’Reilly, *O’Reilly’s Microservices Adoption in 2020 Report Finds that 92% of Organizations are Experiencing Success with Microservices*, `https://www.oreilly.com/pub/pr/3307`, [Online; accessed 16-July-2021], 2020.

[59] oneturkmen, *Miningannotationusagerules*, `https://github.com/ualberta-smr/MiningAnnotationUsageRules`, 2021.

[60] Oracle, *Annotations*, `https://docs.oracle.com/javase/8/docs/technotes/guides/language/annotations.html`, [Online; accessed 13-July-2021], 2021.

[61] Phil Webb, *Spring Boot 1.0 GA Released*, `https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released`, [Online; accessed 16-July-2021].

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

[63] M. I. Rahman, S. Panichella, and D. Taibi, “A curated dataset of microservices-based systems,” *SSSME-2019*, 2019.

[64] Ralph Soika, *RestClientBuilder and jUnit ¿¿ No RestClientBuilderResolver implementation found!* `https://groups.google.com/g/microprofile/c/g73xFffcQNU`, [Online; accessed 16-July-2021], 2018.

[65] M. K. Ramanathan, A. Grama, and S. Jagannathan, “Path-sensitive inference of function precedence protocols,” in *29th International Conference on Software Engineering (ICSE’07)*, IEEE, 2007, pp. 240–250.

[66] ——, “Static specification inference using predicate mining,” *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 123–134, 2007.

[67] Ramesh Fadatare, *10+ Free Open Source Projects Using Spring Boot*, `https://www.javaguides.net/2018/10/free-open-source-projects-using-spring-boot.html`, [Online; accessed 16-July-2021], 2019.

[68] Ratze and rieckpil, *Microprofile JWT web.xml returns 200 instead of 401*, `https://stackoverflow.com/questions/60583971/`, [Online; accessed 16-July-2021], 2020.

[69] Red Hat, Inc, *Red Hat JBoss Enterprise Application Platform (JBoss EAP)*, `https://www.redhat.com/en/technologies/jboss-middleware/application-platform`, [Online; accessed 16-July-2021], 2021.

[70] 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, 2013.

[71] 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.

[72] Stack Exchange Inc, *Questions tagged [microprofile]*, `https://web.archive.org/web/20210627071301/https://stackoverflow.com/questions/tagged/microprofile`, [Online; accessed 16-July-2021], 2021.

[73] L. Tan, Y. Zhou, and Y. Padioleau, "Acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *2011 33rd International Conference on Software Engineering (ICSE)*, IEEE, 2011, pp. 11–20.

[74] The Eclipse Foundation, *Eclipse GlassFish*, `https://projects.eclipse.org/projects/ee4j.glassfish`, [Online; accessed 16-July-2021], 2021.

[75] ——, *Jakarta EE Compatible Products*, `https://jakarta.ee/compatibility/`, [Online; accessed 16-July-2021], 2021.

[76] ——, *Jakarta EE, Cloud Native Java for Enterprise, The Eclipse Foundation*, `https://jakarta.ee/`, [Online; accessed 16-July-2021], 2021.

[77] Thomas Risberg, *Spring Framework 1.0 Final Released*, `https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released`, [Online; accessed 16-July-2021], 2004.

[78] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 283–294. DOI: `10.1109/ASE.2009.72`.

[79] Tom Huston, *What Are Microservices? — API Basics — SmartBear*, `https://smartbear.com/solutions/microservices/`, [Online; accessed 14-July-2021], 2021.

[80] tuesday, *Are there any big spring-boot open source projects?* `https://stackoverflow.com/questions/54782469/are-there-any-big-spring-boot-open-source-projects`, [Online; accessed 16-July-2021], 2019.

[81] user:10747614, *Why Eclipse Glassfish does not support Eclipse Microprofile*, `https://stackoverflow.com/questions/53818892/why-eclipse-glassfish-does-not-support-eclipse-microprofile`, [Online; accessed 16-July-2021], 2021.

[82] VMware, Inc. or its affiliates, *Spring*, `https://spring.io/`, [Online; accessed 16-July-2021], 2021.

[83] ——, *Spring Boot*, `https://spring.io/projects/spring-boot`, [Online; accessed 16-July-2021], 2021.

[84] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, USA: IEEE Computer Society, 2009, pp. 295–306, ISBN: 9780769538914. DOI: 10.1109/ASE.2009.30. [Online]. Available: https://doi.org/10.1109/ASE.2009.30.

[85] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 35–44, ISBN: 9781595938114. DOI: 10.1145/1287624.1287632. [Online]. Available: https://doi.org/10.1145/1287624.1287632.

[86] 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.

[87] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, "Exposing library api misuses via mutation analysis," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 866–877. DOI: 10.1109/ICSE.2019.00093. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00093.

[88] WildFly, *WildFly*, https://www.wildfly.org/, [Online; accessed 16-July-2021], 2021.

[89] Yuriy Bondaruk, *SSL config for outbound connections doesn't work in websphere-liberty 17.0.0.2*, https://stackoverflow.com/questions/45636285, [Online; accessed 16-July-2021], 2017.

[90] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: A study of api misuse on stack overflow," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 886–896.

[91] Y. Zhang, "Checking metadata usage for enterprise applications," PhD thesis, Virginia Tech, 2021.

[92] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 318–343, ISBN: 978-3-642-03013-0.

Appendix A

Supplementary Information for RQ1 Setup

In this appendix, we present a list of sub-APIs we focus on when mining usage patterns in MicroProfile and Spring Boot projects.

For Spring Boot, we treat the following packages as sub-APIs:

- “org.springframework.aop”
- “org.springframework.asm”
- “org.springframework.boot”
- “org.springframework.beans”
- “org.springframework.core”
- “org.springframework.context”
- “org.springframework.expression”
- “org.springframework.jdbc”
- “org.springframework.jms”
- “org.springframework.ldap”
- “org.springframework.messaging”
- “org.springframework.orm”

- “org.springframework.retry”
- “org.springframework.security”
- “org.springframework.shell”
- “org.springframework.tx”
- “org.springframework.remoting”
- “org.springframework.web”
- “org.springframework.http”

For MicroProfile, we treat the following packages as sub-APIs:

- “org.eclipse.microprofile.config”
- “org.eclipse.microprofile.faulttolerance”
- “org.eclipse.microprofile.graphql”
- “org.eclipse.microprofile.health”
- “org.eclipse.microprofile.jwt”
- “org.eclipse.microprofile.metrics”
- “org.eclipse.microprofile.openapi”
- “org.eclipse.microprofile.opentracing”
- “org.eclipse.microprofile.reactive”
- “org.eclipse.microprofile.rest”